PORT LOCALE MODELING AND SCENARIO EVALUATION

IN 3D VIRTUAL ENVIRONMENTS

By

DAMIAN JOHNSON

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 DAMIAN JOHNSON find it satisfactory and recommend that it be accepted.

_____

Lawrence Holder, Ph.D., Chair

_____

Diane Cook, Ph.D.

_____

Robert Lewis, Ph.D.

PORT LOCALE MODELING AND SCENARIO EVALUATION

IN 3D VIRTUAL ENVIRONMENTS

Abstract

by Damian Johnson,
Washington State University
May 2009

Chair: Lawrence Holder

Simulations that are both accurate and detailed can be costly to implement. As such, we examine the trade-offs of using virtual environments provided by the Torque game engine to simulate a real world seaport, including illicit cargo and the sensors designed to detect such cargo. Specifically, we discuss the modeling of daily operations and issues involved in simulating a dynamic environment (e.g, ships, cargo containers, cranes, and trucks). Finally we cover conclusions drawn from using game engines for simulation purposes, both advantages and an investigation of feasibility concerns. Most prominently, issues include the ability to render large areas and the scalability of developing complex interactions. Ultimately, this simulation will serve as a testbed for evaluating optimal sensor deployments and the aggregation of multiple sources of data for decision making.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

From planning for an upcoming meeting to scaled models of architectural wonders, we depend on simulations to provide cheap, yet decently accurate results of fictitious scenarios and to avoid costly blunders. For many environments, such as airports, important highways, and seaports, simulations are more than just a good idea – they are vital. Any interruption can be costly and have wide spread repercussions for users depending on the service. Hence, providing simulations for these sites is necessary to investigate substantial changes that might otherwise require downtime.

In addition to providing an alternative to real-world examination, simulations allow for the use of automated tools and scripts to examine the virtual world in ways that would be impossible otherwise. In particular this allows for the reconfiguration of scenarios to optimize specific concerns. For instance, while rearranging cargo in a seaport to minimize blind spots might mean weeks of lost productivity as units are moved and sensors tested, a simulation could construct and evaluate the scenario in a matter of seconds. Digitalization also opens the doors to the use of machine learning to predict complicated behavior, such as the signals that illicit cargo is about to be smuggled through a port.

Simulations also allow for the examination of complex scenarios that might be impossible or impractical in the real world. For instance during the Cold War the impact of nuclear weapons on cities required millions in the construction of artificial ghost towns that could be monitored as

they were destroyed. Obviously this is a very high price to investigate what-if scenarios, and one that sea ports and other critical infrastructure cannot afford. Simulations, however, allow for in depth examinations of difficult or destructive phenomena limited only by the fidelity of the physics simulation.

Finally, computerized simulations allow for the visualization of data aggregated from multiple sources. Take for instance EPSS [8], a simulation by the institute of Geodesy and Navigation. This uses hundreds of models to depict airports and urban environments (see Figure 1). The depictions are based on real-time data collected from second generation global navigation satellite systems (GNSS), weather, and other orbital data [8].

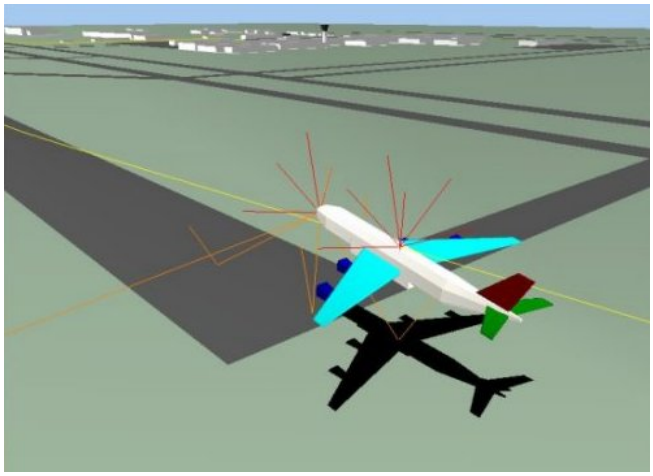

Figure 1: Airport simulation taking advantage of satellite positioning data [8]

The goal of our project is the investigation of the feasibility and related issues in the production of a simulation via a first person shooter game engine. In particular we are interested in the analysis of a real world location for practical concerns. We chose the smuggling of radiological devices through the Port of Seattle, but this could be modified and extended to suit a wide range of locales and scenarios.

We begin with an overview of the project's goals and the testbed environment we are trying to build. Then in chapter three we discuss the game development pipeline and the steps taken in adding scriptable objects to the simulations. Finally we discuss performance and the

engine's strengths and limitations. We then end with related work, conclusions, and future extensions in a continuing effort to use the simulation to draw conclusions about the real world.

CHAPTER TWO

TESTBED ENVIRONMENT

Our research concerns the use of a FPS (first person shooter) game engine to provide an environment in which we can develop simulations for specific locales. In particular we are interested in the process involved in simulating real-world environments and the trade offs of a game engine's use.

Figure 2: Port of Seattle [14]

While there has been much work in simulation development (see related works) the use of a game engine provides a framework with a sense of space and real time which we hope will ease development. In this chapter we discuss the process we took in developing our simulation and the trade offs between ease of development and simulation fidelity.

Our work was done in the Torque game engine [6], a cross platform, proprietary first person shooter engine developed for the 2001 release of Tribes 2 by Dynamix. Since then Garage Games has made a modified version of the engine licensable (with engine source) to independent and professional developers.

The scenario we are interested in concerns the smuggling of radiological devices (nuclear armaments or dirty bombs) through a sea port via cargo freight [7]. For the locale we chose a real location – the Port of Seattle (see Figure 2), the fifth busiest port in North America [14]. We used satellite imagery and aerial photos to determine its layout and estimate operations.

The scenario's faithfulness to reflecting the real world is important. Each step of development used information about real seaports wherever possible, from high definition photos for modeling machinery to satellite imagery for the port layout. However, the primary focus of our research concerned with the process, feasibility, and other development aspects of simulating activity at a real world location via a game engine. More specifically we would like to answer the following questions:

a) How effectively is a game engine, particularly Torque, simulation purposes?
b) What is the process for making a simulation and what issues arise when doing so?
c) How feasible (in terms of persons, experience, and time) is simulation development with game engines?
d) How large of a location can we simulate and at what level of complexity in terms of activities?
e) What are the tradeoffs between realism and performance in simulating a port environment?

Perfectly modeling the full operations of an actual seaport with with everything from the seagulls

to the port workers would be a gargantuan undertaking. Hence we simplify the target scenario to only include the components vital to moving cargo throughout the port, namely the vehicles used in its transportation (ships, cranes, trucks, lifters, etc). While the inclusion of people and the finer points of the port's architecture (light poles, building interiors, etc) would certainly benefit both the accuracy of the results and believability of our simulation, we did not believe they contributed a significant enough benefit to be worth the time of development.

Like most work places there are a number of processes that occur at a seaport. Storage, inspections, changes in crew shifts, and third party work hardly scratch the surface of the potential activities in a given day. However, as we will see later the use of game engines introduces some difficulty into the modeling of complex interactions, particularly arbitrary (unintended) behavior. Hence for this initial testbed we simply focused on the transit of cargo from a ship throughout the seaport.

CHAPTER THREE

GAME DEVELOPMENT PIPELINE

Integral to the project is the environment in which the simulation takes place. Common operations such as the loading/unloading and transportation of cargo need to be replicated with a fairly high degree of fidelity for any conclusions to be drawn. For our simulation we have decided to focus on operations performed by ships, cranes, yard hustlers (trucks), pickers, and trains in their movement of cargo.

Conventionally game development works like a pipeline. Modelers provide their creations to texture artists, who add surface details and hand the results to the developers, who integrate them into the game. Few people have the cross disciplinary skills needed to work on multiple aspects of development and this allows them to focus on the parts they can do the best.

The ETC (Entertainment Technology Center) at Carnegie Mellon University, for instance, splits students into development teams of four with a developer, modeler, texture artist, and audio engineer [18]. After the initial conceptual design they go about making the art and framework needed for their game, closely collaborating and passing work among themselves in this fashion.

Even the development guide for the game engine we are using, Torque, suggests having a small team with work split among an artist, programmer, and designer. Our project was not able to afford such luxuries, rather necessitating that each contributor was a jack of all trades. Still, the addition of interactive objects into the simulation fell nicely into the pattern used by the ETC: modeling, texturing, and finally scripting the objects functions.

3.1 MODELING

A substantial amount of time, if not the majority, was spent modeling the things the simulation would hold. Detailed models are not strictly necessary – as far as the engine is concerned all that matters is the collision mesh so a boat could just as well be a collection of cubes. However, a world of boxes would not only be unintuitive and confusing, but make the demonstrations far less convincing to a lay audience (port personnel not familiar with abstract representations found in computer science). Understandability was one of the primary reasons for developing a real-time simulation so this is very important.

The question of just how much detail is sufficient for port personnel to understand the simulation is subjective, yet important. Some individuals might be able to see past substantial

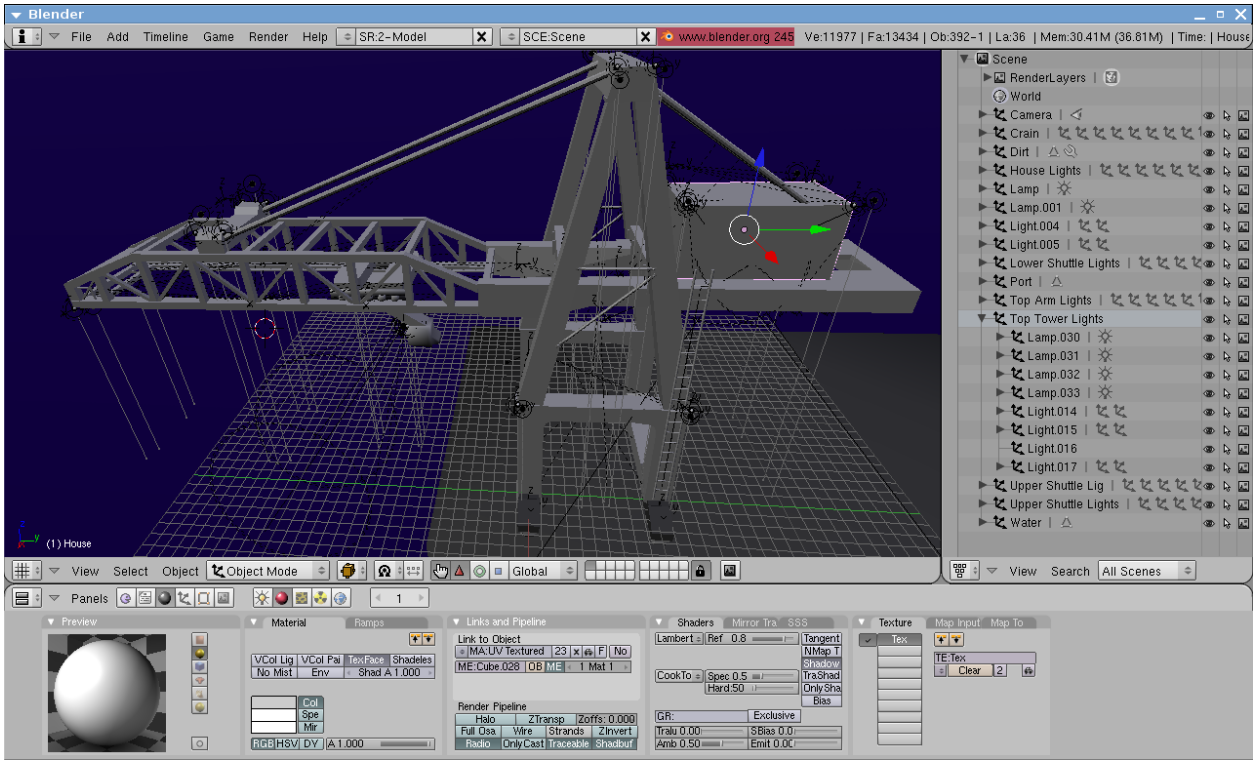

Figure 3: Blender screenshot while modeling crane

breaks from realism while others would find them distracting. Details tend to be easier to remove than add, so our aim has been to provide the most visual fidelity we could easily include. This will provide a point of contrast (against its scaled back counterpart) for future usability interviews and HCI (human computer interaction) studies.

The simulation calls for several interactive models, primarily vehicles. To develop them we used Blender [2], a prominent Open Source 3D modeling tool that has been used for several high-end works like Elephant Dreams and professionally in Spider-Man 2. This offered an efficient interface for quickly sculpting the various parts we needed (see Figure 3). Blender also provides excellent UV texture mapping support and a level of detail far higher than what our game engine could support. Developing models with an unusably high level of detail is conventional in the game industry for a couple reasons – first it is easier to subtract details than it is to add them. Second, this allows future releases to take advantage of advances in model rendering without redoing the model (ie, easily improved graphics).

Most things in the world can be first broken down into the basic geometric shapes that constitute it (cubes, cylinders, spheres, etc). This is especially true for most man-made objects like the vehicles that needed to be developed. With the truck, for instance, tires began as cylinders and the body a series of cubes. After the shape has been added and scaled to the proper proportions in comparison to the rest of the figure it is deformed, vertices moved and faces extended to reshape the figure to fit the

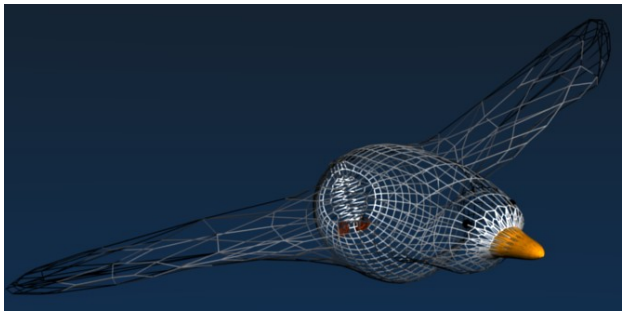

Figure 4: Wire frame of the seagull model

desired part with a higher degree of detail. For instance the body of the seagull (see Figure 4) began as a cylinder, then had faces extruded, curved, and molded to provide a figure fitting a bird.

When a part is finished it, is added to an organized hierarchy. For instance a window belongs to the door to which it is attached. Besides making object navigation (in what often becomes a sea of hundreds of parts) easier, this lets us independently transform larger sections of the figure uniformly. For instance with a single scaling operation a truck's cab can be made to better fit the rest of the model.

The organization also allows for easy duplication. Much as programmers reuses code, modelers reuses the models they make. For instance only half of the truck is actually modeled – the other side is simply a mirror reversal letting us avoid hours of detailed, error prone work. The final benefit is for animation – most game engines allow for transitions between a series of key frames that either loop or occur on demand. For instance, this might be done for a dog's wagging tale or the spinning of tires. This was done for the only non-mechanical model: a seagull that flaps its wings. The motion was done via armatures, bones that move the vertices around them. Though the model is done this one has not been exported to the engine yet (a low priority since sea birds are purely aesthetic).

Lastly objects are textured, primarily using UV mapping. The 'UV' represents spacial coordinates, the method being a means of mapping 2D textures to a 3D surface. In practice this means finding an appropriate texture, shifting the 2D representations of the vertices so the texture has the desired mapping, then occasionally painting the texture to get the desired

appearance along edges. This produces an intuitive means of getting a high degree of detail, and if done right can give a more three dimensional feel without needing to explicitly craft additional details.

We desired the most realistic feel from the models we could provide so a large part of the time was spent studying a series of high definition photos to pick up the intricacies involved in everything we made. Since the models are three dimensional we needed images from several angles, and a bit of guesswork for blind spots and missing components. If the resolution is high enough the photos can double for textures, allowing a more accurate depiction of the object we are trying to duplicate. Images with multiple things we want to model can be especially handy since they provide a relative sense of scale.

Ideally this process would include first hand access to the vehicle being modeled so high definition images could be taken from all angles (including hidden surfaces such as the bottom and interiors). Also, if measurements of all objects in the simulation were known, then scaling could perfectly reflect an object's actual relative size. However, we would probably want to enlist the aid of a professional modeling artists before attempting to include such a high level of detail. Also, as we will mention later, much of the finer points of these models would be lost in the DTS export and also add a significant graphical burden.

Another possible improvement to this process would be to use procedural modeling, which is the use of algorithms in defining the scene. The term covers quite a few forms of automation, from texture generation to model deformations (such as weathering). However, for our simulation we are chiefly concerned with its uses for model generation (either through

recombination of constituent components or using a special grammar) and mutation (changing the model according to given input parameters). Blender provides support for this via Python scripts that make use of its internal "bpython" API [2].

When modeling a crowd assigning features to each individual is unnecessary. Procedural modeling can be used to provide variance between the individuals, using different texturing and dimensions for each [11]. This technique could be used to ease the burden of creating port personnel and allow our simulation to be manned by an arbitrary number of dynamically created (unique) individuals. In addition through generative modeling some portions of the scene, such as buildings, can be defined via a shape grammar to be entirely generated procedurally [12].

Since subsurfacing (smoothing calculations) are stripped from exported models quite a large number of vertices are used in all curved faces. This causes models to have a decent appearance within the game but places an added graphical burden we might not be able to afford. One alternative is to use procedural modeling to generate models on the fly with vertex counts based on the system resources available. This would use a base model and given detail level to subdivide curvatures on demand [11].

The level of detail we can render is both hardware bound and dependent on the scene being simulated. With a quick initialization test Torque could be asked to render a series of models and determine the relative frame rates. This would provide a platform dependent measure of the graphical load we could accommodate, then generate models with an appropriate level of detail. Torque could also be configured to take frame rate samples while running to scale back graphical quality when performance drops below a given threshold.

This work, being firmly planted in the computer science department, meant we did not have access to artists. Fortunately the replication of machinery takes more architectural than artistic skills, so even in the hands of developers they turned out decently well. Source models for the various finished components can be found at:

http://www.eecs.wsu.edu/sgl/portSim_modeling.php

3.2 DTS EXPORT

The DTS file format, which stands for Dynamix Threespace Shape, is the primary format used by the Torque game engine (along with the DIF format that includes pre-calculated lighting for static objects). Being proprietary this is not natively supported by 3D modeling tools. Fortunately, plugins exist for several popular modelers (including Milkshape 3D, Maya and Blender) capable of exporting conversions.

In general the exporter respects the structure and textures provided via UV mappings. However, the DTS format unfortunately introduces several limitations that have an obvious impact on the quality of the converted models. For instance the translation strips lighting, sub-surfacing (which is used for smoothed surfaces), multi-textured faces, reflections, greatly diminishes texture fidelity, and has limited support for transparency. Most noticeably the engine uses an excessively high degree of ambient lighting, bleaching the result. The aesthetic loss is by no means subtle and greatly reduces the realism of the the resulting simulation (see Figure 5).

In addition, the format has poorly documented rules that need to be followed for exported files to function. Common pitfalls include:

1. Texture dimensions must be a power of two. This used to be a common requirement to aid in hardware rendering, but since has been lifted in more modern engines.
2. UV mapped images are not integrated into the exported file, so they need to accompany the DTS model in the same directory.
3. Textures require proper flags, for instance a transparency flag if a material contains alpha

(opacity) values.

4. The DTS format relies on metadata for the engine, often relayed to the exporter by hacks like added invisible marker objects.

Mistakes silently fail, simply making any surfaces with a problematic or missing textures a light gray. Since this does not show up until its use, this can easily mean hours of guess-and-check work to track down problematic details.

The only aspect of the models that has meaning to the engine is the collision mesh. It is provided via separately parented shapes that roughly conform to the shape of the figure. This can be a transparent copy of the model if a perfect collision mesh is desired, but this can be costly for the engine, meaning extra collision calculations when players get within the shape's bounding box (the furthest most reaches of the shape). Hence it is advantageous if the mesh is simplified in areas where detail does not matter, especially rounded sections that might contain hundreds of vertices. This mesh only has meaning if added to the scene as a static shape (i.e. one that cannot move), and even then only seems to be considered for the player character. AI players simplify movement calculations by considering the bounding box instead.

The burden a model places on the engine is not a factor of its size or textures, but rather the number of vertices. This is not an immediately obvious attribute to the human eye, to which a cube is just as complex as a cylinder. Hence it is often a surprise when a deceptively simple figure like the truck holds the means of bringing the engine to a crawl (due to several rounded surfaces).

Figure 5: Demonstration of fidelity loss between Blender rendering (top) and DTS export (bottom)

3.3 SCRIPTING

After we have finished with the DTS model export, the models can be added to the scene as static shapes. However, if the object is to do something, then it will need some logic. All activity was first developed in a simple demo scene (see Figure 6) that just contained the objects being worked with. This lightweight environment was used to simplify the writing and testing of scripts for new or experimental features.

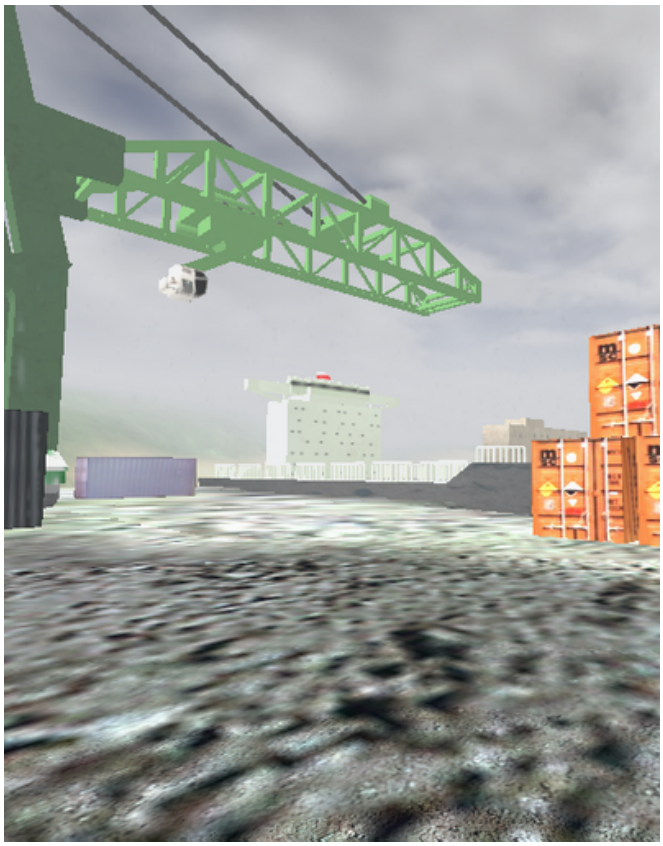

Figure 6: Demo Coast Scene

To be flexible and ease development almost all game engines handle this by scripting, often Lua [1] or using another established scripting language. This is vital since it allows easy modularization of game development, between the scene logic and the engine that supports it. Scripting languages use callbacks provided from the engine (usually C/C++) to trigger functionality that is either internal to the engine or requires substantial computational muscle. In addition to providing clear separation of functionality a scripting language provides all the ease of using interpreted languages (memory abstraction, garbage

collection, easier syntax, etc.) for any code that is not performance critical (the vast majority).

In our case Torque uses a proprietary scripting language named, simply enough, Torque Script. As a language it is dynamic with a vaguely C-style syntax (leaning a bit toward PHP). It is also type insensitive, treating *1 == 1* the same as *1 == "1"*. The language does not support distinct classes for collections, instead treating everything internally as a string. Instead of common list operations (add, remove, get, etc.) the programmer needs to use text operations (such as substring) to emulate the functionality. Like many dynamically typed languages it also defaults variables to empty strings, making bugs derived from typos especially difficult to spot.

Between a strange syntax and fairly quiet error handling this is not a terribly easy language to code in. However, the largest difficulty is the lack of documentation – the standard library and callbacks lack even basic explanations of their arguments or examples of use. This is a well known deficiency of Torque and one Garage Games claims to be addressing. For now, however, most development consists of guess-and-check calls that can make even simple script writing eat hours if not days.

On top of this Torque, as with many game engines, is not particularly stable nor fully implemented. The game engine was designed to support first person shooters and hence those derived use cases were the only ones to be fully developed and tested. Anything else is hit and miss. For instance, item instances can have forces applied to them in bursts via a function called 'applyImpulse'. This allows us to push them about the scene manually via scripts. However, while instances of AIPlayers have this function as well, it is not implemented (a no-op), forcing us to find other means of providing manual movement.

Often the lack of documentation and bugginess leads to the perfect storm of unusability. Take the hovercraft for instance. This seemed the perfect representation for boats – it maintains a fixed distance from the ground (which can coincide with the surface of the water) and can be 'driven' if we ever made the simulation interactive. However, there is a surprising deficiency of example usage with hours of research only yielding instances several years old (and incompatible to our version of the engine), and documentation gives little more than the names of the variables being used. Using hovercrafts would still be doable if only by guess-and-check experimentation with the variables except for one major issue: misconfiguration results in the game engine freezing the system with substantial fallout. This collateral damage comes in the form of mouse functionality. The game engine holds mouse focus at the time of its freezing so when a kill command is issued Torque takes mouse controls to the grave with it. This means a full UI manager restart (Gnome in this case) is needed before we can hazard another guess at the variables used by hovercrafts (unfortunately modprobe and other driver reloading were unable to regain mouse functionality).

This said, Torque comes with demo environments and example scripts, one of which became invaluable to our simulation: a path following AI player. This script allowed us to litter bread crumbs (markers) across the scene that the AI can be told to follow. This is what we ended up using for all vehicle movement, from boats coming into the port to trucks driving the cargo away.

The engine code that handles AI player movement has not been configured for three dimensions and when the player loses contact with the ground the item freezes, refusing to move

toward the next marker until it is in contact with the ground once more. Despite altering the movement engine code to properly handle three dimensional movement, somewhere within the deep hierarchy of inheritance a parent class seemed to prevent this functionality (fixable, but the offending code is not easily found).

To allow us to lift and transfer cargo we applied a hack – the AI player is told to ignore gravity (by setting the gravimetric force and air resistance to zero) and the velocity is set via scripted means. When the target location is reached velocities are set to zero so it will hover in mid air. With a series of three velocity setting/stopping operations we are able to emulate the transfer of cargo from the ships to the trucks and vice versa.

The final piece of scripting voodoo in our scenario is the fluent transfer to multiple trucks. This is largely handled by triggering events on either arrivals or collisions, common callbacks available to our vehicles. The scene proceeds as follows:

1. Initially a single truck is spawned and moves to one of three locations under the crane. These points lie parallel to the orientation the ship will take as it docks and each lies along side pieces of cargo. When it reaches this point it stops.
2. The ship is spawned with six pieces of cargo configured in a 2x3 pattern on its deck. Some may contain bombs (the threats our scenario is being designed to detect). These bombs are modeled as particle emitters that shoot projectiles in the direction of static sensors. Due to a high rate of fire this emulates a steady emanation of radiation that may or may not be blocked by various obstructions. The details of this are being studied by

another researcher in the group [4].

3. The cargo ship moves into the port and positions itself under the crane. Once it has arrived at its final destination it stops and triggers a script to transfer the cargo.
4. The cargo transferring script disables gravity for the cargo, unmounts them from the boat, and applies the aforementioned sequence of impulses to transfer them to the port. This is done via numerous scheduling calls so cargo will be transferred one at a time.
5. When cargo collides with the waiting truck it is mounted to the back and the truck drives away, following a predefined path. A new truck is spawned and takes position under where the next piece of cargo will fall. As a cargo container carrying an emmitter passes through the port, the proximity and obstructions between the various sensors are used to determine if it is detected.

This has the result of producing a relatively seamless transfer. Currently the only obviously deficiencies lie in the path following. AI players make sharp turns to head toward path markers, giving a very unnatural look to parts of the path. This can be somewhat fixed by adding additional markers to make turns more gradual, but this in turn makes changes to the path later on exceedingly difficult.

Under the covers the vast majority of this scene is hard coded values and positions. For instance placement of the truck markers was refined dozens of times to produce the best visual results. This means modifications to the scene, though not difficult, can be very time consuming.

In addition, AI objects always turn to face the next destination they want to move towards.

With some hacks we managed to make the positions coincide nicely enough that cargo faces the proper direction while in transit, but in the later parts of the scenario (when taken away by trucks) the bug becomes obvious. We have tried clearing path data, using different object representations, etc. but most remedies have drawbacks of their own that make them unusable.

3.4 SUMMARY

Game development is conventionally done in a pipeline fashion. For us this contained three distinct steps: modeling, exporting to DTS models, and finally scripting the logic for our simulation. Both modeling and scripting are substantial development tasks in themselves and comprised the vast majority of time in developing the simulation.

Modeling was done in Blender and included studying the objects being modeled, replicating the basic geometry, refinements to add detail, and finally the application of textures via UV mapping. Models are chiefly for aesthetic appeal, with only their collision mesh holding value to the simulation. Nevertheless, decently realistic models are vital for our simulation to have a believable appearance and make the authenticity of results easier for a lay audience to accept.

DTS exporting is the step of turning the highly detailed Blender model into a format understandable by the game engine. While error prone, once the rules for exporting are understood this simply involves the execution of the exporter script.

Finally, scripts are written to make the object perform as proscribed by the simulation. This is done via a proprietary scripting language that executes hooks made available in the engine. The use of scripting eases development and decouples game logic from the engine running it. However, limited documentation and spotty functionality made this step rather difficult.

CHAPTER FOUR

INTERACTIVE SCENARIO OBJECTS

This section provides a discussion of each interactive (scripted) part of our scenario, discussing how the components interact and individual issues in their development. This will not cover objects crafted for purely aesthetic reasons (like the seagull) nor static shapes (pieces that do not have scripted logic for the scene such as the crane and lifter). Rather, this will focus on the parts currently involved in cargo transit: ships, trucks, and the cargo itself.

The crane and lifter are currently rendered as static shapes for different reasons, both of which are not likely to hold true in the future. The crane's function at the port is simply to lift cargo off docked vessels. Hence its arm should pivot among cargo as they are being unloaded. However, this is not especially easy functionality to script and holds a limited value to the realism of the simulation, so it was not implemented. As for the lifter, the current scenario simply transports cargo directly to waiting trucks rather than one of the temporary holding areas, meaning that its services were not required.

4.1 FREIGHT

In its simplest (static) form cargo is simply a textured box. Eight vertices with textures associated with each side. However, the pieces of cargo that might contain bombs were made to be three dimensional (sixteen vertices to include an interior). A mount point (location where another object can be attached) was added to the center so it could contain bombs or other contraband.

Since Torque is not able to handle multi-textured objects, all the cargo's constituent parts were included in a single image (cookie-cutter fashion, see Figure 7). The UV mapping then associated each face of the cargo with the appropriate part of the image.

Figure 7: Texture used in UV mapping of cargo.

Unlike vehicles the cargo never needs to follow path markers or travel along the ground. However, they still needed to be scripted as AI Players so that collisions with other objects (such as ships) can be detected. This might be changed in the future (via engine modifications) as a possible solution to having cargo face the wrong directions (see Section 3.3 for a discussion of this deficiency).

Cargo is currently the most numerous type of item in the port. As we will see later the overhead of rendering each individual item (despite having a low vertex count) becomes an issue once they number in the hundreds, which is typical for a real port. To solve this we consolidate

large blocks of cargo, that will never need to move, into a single hollow brick (see Figure 8). In other words the cargo model is duplicated in Blender and the internal vertices and faces (which are never visible) are removed. This will have a negative impact on the realism for particle physics since radiation passing through the brick will no longer encounter numerous faces. This might be fixed by using special materials (i.e., the brick surface counts as having the properties of several individual walls), however we decided the effect on performance and the complexities of populating the map were not worth the small effect on realism.

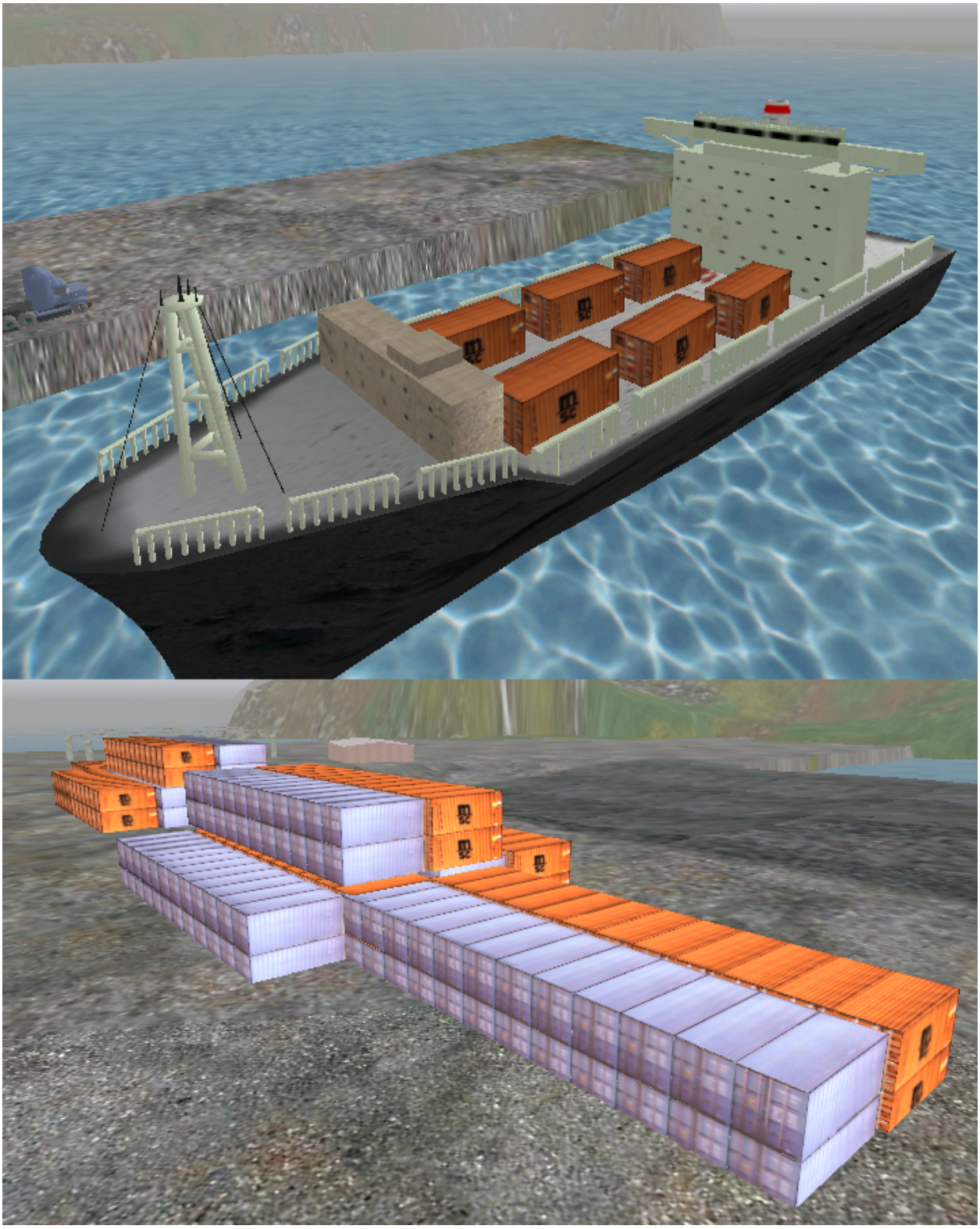

Figure 8: Mobile cargo being brought into the port (top) and static bricks (bottom)

4.2 CARGO SHIP

Cargo ships were based on a Wikimedia Commons high definition photo of a Hanjin container ship leaving the Port of Seattle. Since this was among the first models done, the texturing and level of detail is noticeably lacking in comparison to other models. For instance windows in the ship's tower were done via textures (black dots) rather than being built into the model (see Figure 9). This gives an unreal, 2D appearance. However, for performance this turned out to be inadvertently advantageous (see Section 5.1).

Currently the sole purpose of ships for our simulation is to bring cargo into the port. When the scenario starts, a ship is spawned in the distance with several pieces of cargo attached. It then maneuvers into position under the crane, stops, and triggers an event telling the cargo to get off. Torque limits the maximum number of mount points on an object to eight, so we currently cannot accommodate the hundreds of pieces of cargo usually found on container ships.

The only point of interest for the ship model was how it was made to 'float' on the water. For reasons mentioned in the scripting section we wanted to avoid the use of specially scripted hover vehicles. Hence we used a hack in the DTS exporter. Objects appear relative to the center of the 3D modeling tool's scene, so to make an object appear to 'fly' it simply needs to be lifted before being exported. This way, though the engine still thinks ships are driving along the ground it will *appear* to be floating in midair (at a height which coincides with the surface of the water). Finally we flattened the seabed under the route ships travel so they move evenly over the surface.

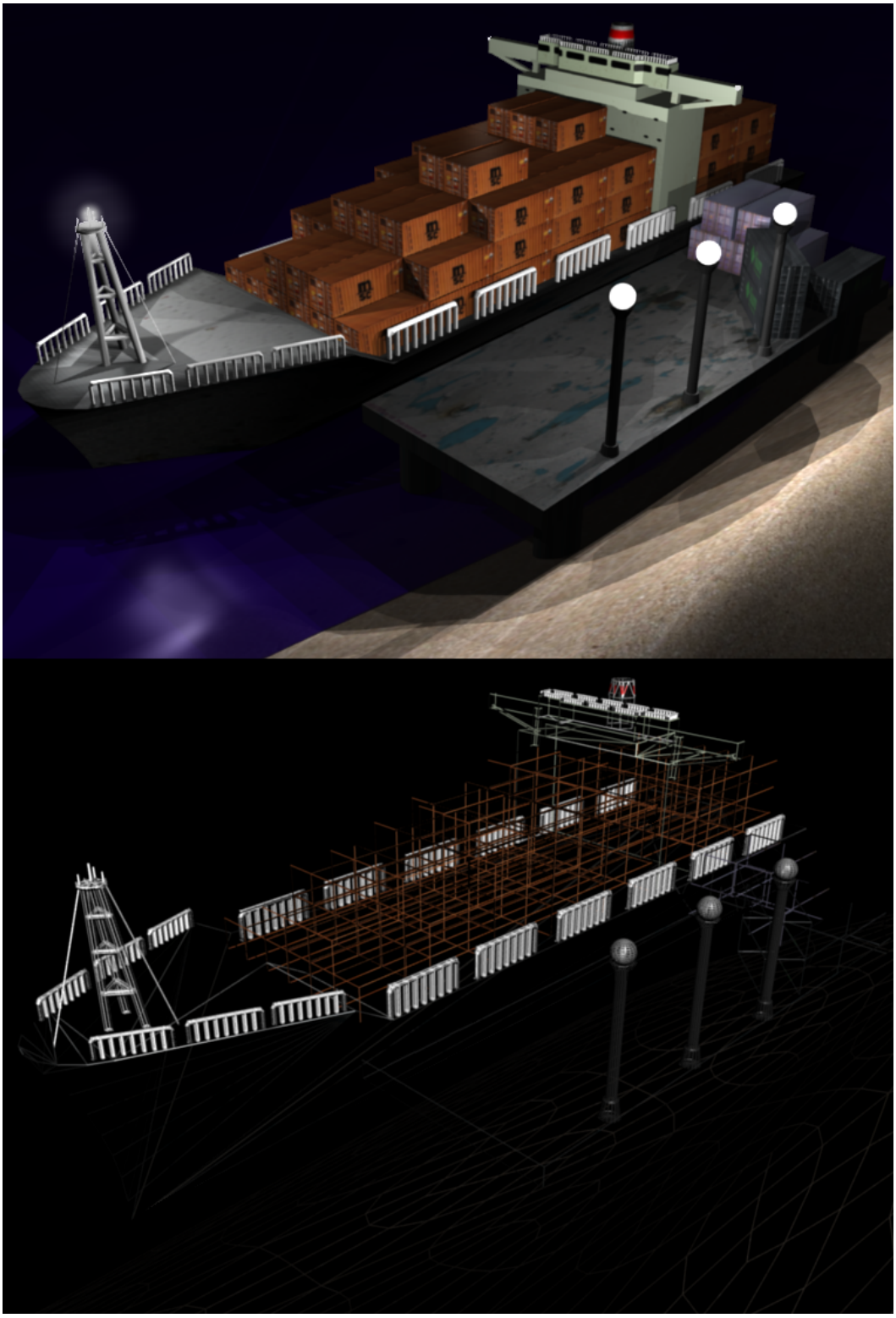

Figure 9: Blender rendering of ship model used in simulation (top) and wire frame counterpart with 10219 vertices, not counting the cargo and scene (bottom)

4.3 FREIGHT TRUCK

The final (and most complicated) model was that of the freight trucks. This was based on dozens of high and low definition truck images to most closely approximate the appearance of each component. No single image provided a view of all sides in enough detail, so the final result was a composite of several (similar) models of trucks. As mentioned in Section 5.1 the final model was prohibitively difficult for the game engine to render (see Figure 10), so parts of the geometry were simplified. One very simple fix was to remove the UV spheres being used for bolts on the tires. UV spheres have particularly heavy geometry, consisting of 242 vertices apiece. With eight per tire on nine tires that marks a decrease of 17,424 vertices without even being noticeable to the user (over halving the graphical requirements). Further improvements are not likely to be as transparent.

In our scenario trucks spawn, drive under the crane, then wait for cargo to land on the truck's bed. AI players (like the truck) detect collisions based on scripted bounds which, through numerous iterations of guess-and-check, roughly approximated to the height of the rear platform. When a piece of cargo touches the truck, it is attached to a mount point and driven away, following predefined path markers throughout the port.

Figure 10: Blender truck rendering (top) and wire frame counterpart with 27334 vertices (bottom)

4.4 SUMMARY

In this chapter we discussed the three models of primary importance in our current scenario (shuttling cargo from a ship through the port). Cargo consists of two types: static and dynamic. Static cargo simply provides obstructions, reflecting the stacks of freight that usually litter seaports. Dynamic cargo has an interior and might or might not contain contraband. It has precious little logic of its own, usually being mounted to a mode of transportation.

Both ships and trucks function in a similar fashion: when cargo touches them it is attached and the vehicle carries it along a predefined path. This path is defined by markers placed throughout the scene which can be easily adjusted to reflect new routes through the port. Under the covers much of the logic used to make the scene function is non-obvious, consisting of little hacks to make the engine behave as desired.

CHAPTER FIVE

RESULTS

Real world environments are large, and while we have a working demonstration for the offloading of cargo from a single ship the question has yet to be answered – how does this scale? Both in terms of computation and development we need to be able to simulate large areas with complex and intricate interactions. So how does the environment fare when the need for expansion arises?

The answer, given the current engine and tools, is 'poorly' though not impossible. As we will see, detail, either graphical or in terms of realism for arbitrary interactions, can become costly. The level of graphical detail, especially within the purview of the player, quickly leads to noticeable performance hits that limit the number of objects we can have in the scene.

In addition, the use of game engines as the basis of the simulation brings both advantages and disadvantages as we will discuss. Realism, particularly in scripting arbitrary interactions quickly becomes time consuming and brittle. First person shooter engines in particular were not designed for complex interactions like what we would want in a port. Hence while hard coding positions is quick and perfectly appropriate for game engines, it makes refactoring environments with dozens, if not hundreds of delicately interacting parts very complex.

5.1 RUNTIME PERFORMANCE

When it comes to simulating a large environment like the Port of Seattle the issue of computational load quickly becomes a point of interest. How much of the port can we feasibly simulate and at what point do we start seeing a performance hit?

As with most games the majority of the processing rests with the graphics. Torque is hardware accelerated, but still almost any scripts we write cannot compare to the demands of rendering complex geometry. The one exception to this is the particle physics that go into representing radiation, especially in complex interactions when they collide with various material. Section 5.2 describes the performance impact when particles are introduced into the simulation.

The models used by our simulation were developed over the span of two years, the first naturally being fairly amateurish and growing in complexity. Table 1 shows the statistics of each (ordered by complexity which roughly matches their chronological order, newest to oldest):

	Vertices	Faces	Objects	Model Size	Textures Size
Truck	27334	31930	163	3.7 MB	14.4 KB
Lifter	24554	27032	205	3.3 MB	530 KB
Ship	10219	11731	419	2.2 MB	3.2 MB
Crane	7302	8464	307	1.7 MB	3.2 MB
Seagull	2198	2206	3	225 KB	491 KB
Cargo	8	6	1	122 KB	104 KB

Table 1: Model statistics (vertex, face and object counts do not include hidden collision meshes).

Ignoring the seagull and cargo (which pale in complexity compared to the models for port

machinery) something of interest to note is the trend over time for vertices, face counts, and model sizes to grow. The reason for this is simple: over time models grew in detail with the developer's skill, allowing figures to be broken into smaller and smaller constituent parts, providing a better level of realism. We later discovered this to have a substantial performance impact as we will see later. Still as a basis this is advantageous since it is far easier to remove detail from a figure than it is to add.

To a large extent vertex counts are dictated by the figures being modeled. Any sort of rectangular shape constitutes few vertices. For instance a piece of cargo is a block, and hence eight vertices. Unless handles, an interior, or some other level of detail is added, any additional vertices are a waste. A tire, however, is in essence a deformed cylinder meaning that even in its simplest form (a plain cylinder) it might require dozens of vertices to avoid having an obviously blocky shape. Then, when details are added such as groves, bolts (UV spheres which inherently have a *very* high count), etc. this can easily grow to be hundreds, if not thousands of individual vertices.

The other obvious trend is for the cumulative texture sizes to get smaller. This started with figures using multiple megabytes of detail and shrinking until complex vehicles were wrapped in only a dozen kilobytes of images. This too is a result of becoming a better modeler, using smaller textures more efficiently without a loss of detail. The assumption behind this was that wasteful texturing would eat a substantial amount of video memory, causing a performance hit if this expanded to require swap. However, as we will see texture sizes turned out to be meaningless for simulation performance, making this effort largely a loss.

The object counts in all cases are based on the thing being modeled and did not substantially impact performance. For instance, the ship had the most vertices because of the numerous hand rails, each bar of which was a separate, low-detail cylinder.

For these experiments we need a metric that reflects the load as we tax the system with models and particles. The computer itself offers several attributes that would reflect burden on the system, such as CPU, GPU, and memory usage. Other alternatives could get even more indirect such as the system temperature as the burden grows. However we propose using a far simpler (and more conventional) measure.

Torque has not been designed to take precautions to preserve game performance. It will not drop particles or crop models as the burden becomes unmanageable. Rather, the only best effort service the engine offers is fluent visuals. When an impossibly large number of particles need to be tracked Torque does so, at the cost of frame rate. This is to our advantage because it gives us a metric indicative of the system's load. This an easy measurement to take, in addition to being fairly deterministic on the system (reproduce able) and a performance indicator of obvious importance to the user.

Other proposed measures such as polygons, faces, vertices, game cycles, and particles per second are a measure of the system's capabilities (i.e., how much of these items can be accommodated). These are interesting measures because they give us an idea of both the extents of what we can process and a relative measure of how these attributes compare to each other (i.e., roughly how many particles equal rendering a vertex). We will be using the engine's decline in frame rate (symptomatic of a rise in load) to indicate how these affect performance.

Rendering time is based on several factors. In its general case it can be expressed by the function:

t = RT (SG, RA, HW, ST)

where SG is the scene graph, RA is the rendering actions used to generated a 2D view, HW is the hardware, and ST is the state of the system [20]. We simplify this by replacing the first two parameters (SG and RA) with the viewable objects, attributes, and geometry (since the RA is fixed by the engine). The rest of this section focuses on the first attribute (being the most relevant to simulation development). Investigating the effects of hardware was proposed but dropped in favor of focusing on simulation attributes. Hardware and state, being important factors of the rendering time, are discussed in future works.

To check performance we used FRAPS [5], a tool that provides video and screen captures, along with a measurement for a game's frame rate. No special precautions were taken to lighten the system's load since we were only interested in rough approximations of the relative load provided when rendering various models, but the specs are:

OS: Windows XP (SP2)

Processor: Intel Core 2 (6600 @ 2.4 GHz with 2 GB of RAM)

Graphics Card: Nvidia GeForce 7900 GS

Torque Version: 1.5.2

As with most engines Torque simplifies rendering to only things within the player's field of view.

Hence the load without model rendering (best case performance) can be found by simply looking away, such as at the ground. Worst case is felt when watching the entire scene, which is most easily done by taking an aerial view.

In the aforementioned best case we found the game engine runs at 64 frames per second.

	1	2	3	4	5	6	7	8	9	10	15
Truck	64	32	21	21	16	16	13	11	10	9	6
Lifter	64	32	32	21	16	16	13	13	11	10	6
Ship	64	64	64	33	32	32	32	32	21	21	16
Crane	64	64	64	64	33	32	32	32	32	32	21
Cargo	64	64	64	64	64	64	64	64	64	64	64
Bomb	64	64	64	64	64	64	64	64	64	64	64
Sensor	64	64	64	64	64	64	64	64	64	64	64

Table 2: Frames Per Second (FPS) / Model Count

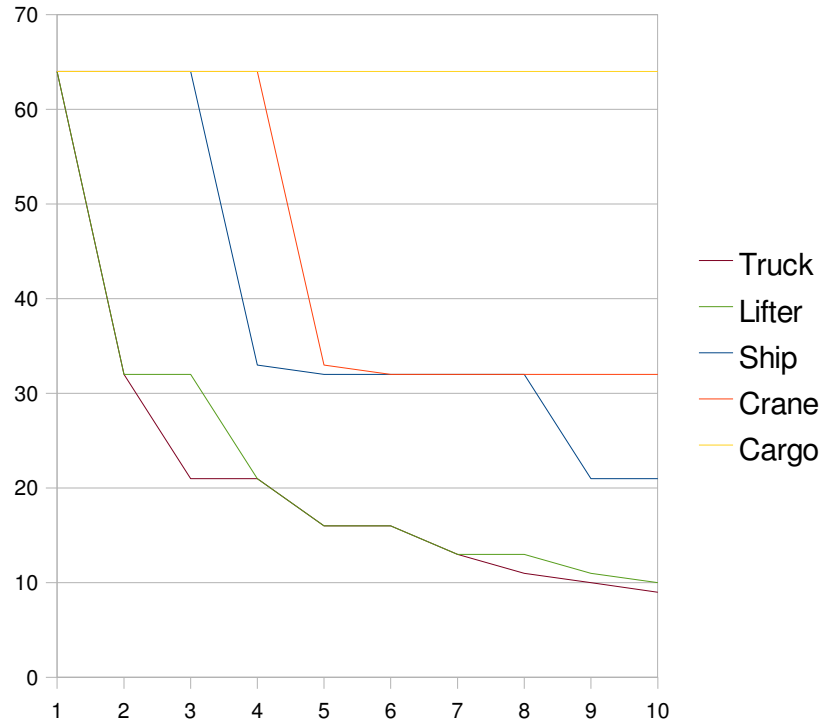

Figure 11: FPS vs Count for DTS Models

These results are interesting since they show a clear, sharp stair step trend toward halving the frame rate as computational demands increase. Thus, it is likely that as long as the graphical demands are kept below a certain threshold, we should see perfect performance, rather than a gradual loss as demands grow.

We later discovered that Torque has an internal means of detecting the frame rate which show a more gradual decline in performance. Without more information on the relative means of measurement we cannot be sure of the reason for this discrepancy. They both achieve similar results as loads rise and for consistency we stick with the FRAPS measurements for these calculations.

The correlation between graphical demands and model complexity (as opposed to, say, texturing memory requirements) seems strong. Table 3 shows the vertex counts when the first drop in frame rates were detected.

	Truck	Lifter	Ship	Crane	Cargo	Bomb	Sensor
Count in frame rate drop	2	2	4	5	340	93	55
Vertices being rendered	54668	49108	40876	36510	2720	---	---

Table 3: Scene Complexity at First Drip in Frame Rate (bombs and sensors lack vertex counts because their original Maya models were lost)

This is interesting since while the model count varies, the vertex counts are roughly similar regardless of the models used, providing further evidence that vertex counts (i.e. model complexity) are the determining factor in game engine performance. More importantly, though, this gives us a system dependent count for the number of vertices we can render before facing a drop in performance. Since the rendering cutoff is almost certainly hardware bound, we can

approximate the cutoff to be around 32768 vertices (2^{15}). The cargo is an obvious anomaly, probably owing the lower performance to needing to render hundreds of individual objects to see its first performance hit (as shown in Figure 12 this is a significant number of individual objects to track). This added overhead for individual objects may become a concern as the simulation grows to include more and more constituent parts (Section 4.1 discusses one possible solution).

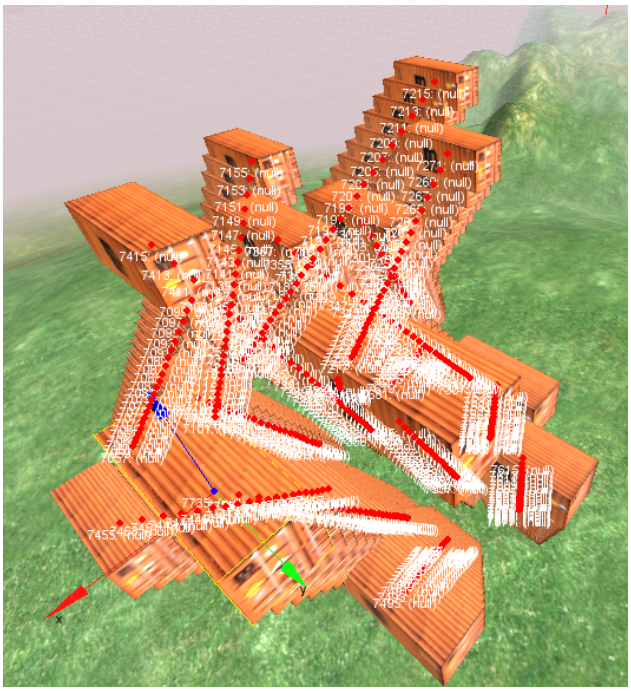

Figure 12: Engine rendering 340 pieces of cargo (resulting in the first sign of slowdown)

Before discovering these results, efforts to improve performance primarily focused on two things:

1. Reducing the memory used by textures, the reasoning being mentioned earlier.
2. Reducing the number of objects being rendered. This was mostly done by making a model for a large, hollow cluster of cargo. This removed the need to have interior faces and greatly dropping the number of static shapes in the simulation.

The results mentioned earlier clearly show neither of these efforts made an impact on the performance of the simulation. Most likely Torque or the DTS export is limiting textures to a lower resolution with a fixed relative DPI (hence why texture sizes seem unrelated to

performance). Rather, what we should focus on is dropping the vertex counts, either by limiting the number of complex shapes (like trucks and lifters) or editing the models to remove detail and dropping vertices from curved surfaces. This will give objects a blocky appearance, but would be an easy way to allow for a far larger simulation.

One final note- the above results were provided from the use of static shapes, which have a simpler representation in the engine since they never need to move. When rendering dynamic trucks the loss in performance was roughly double that of their static counterparts, which may be problematic if the simulation complexity grows to include more activity (and by extension dynamic shapes) in the port.

5.2 PARTICLE PERFORMANCE

Our simulation does not exist in a bubble. Its goal is to test the ability of ports to detect radiological threats given specific sensor placements. So how does it perform as these demands are varied? There is not anything special about particles firing, per say, we are simply interested in how a computationally intensive task works along side the simulation, particularly when model rendering becomes demanding.

First we need to figure out the load placed on the engine by particle emissions and what varies these demands. Particles have a few attributes: fire rate, longevity, and collisions. Longevity is only of concern in that it helps dictate the number of particles present at any point in the scene. Again we will use FRAPS to measure performance in frames per a second as these parameters change. All of the following tests we had ten sensors circling an emitter in empty space to keep the rendering demands from interfering with the results (see Figure 13).

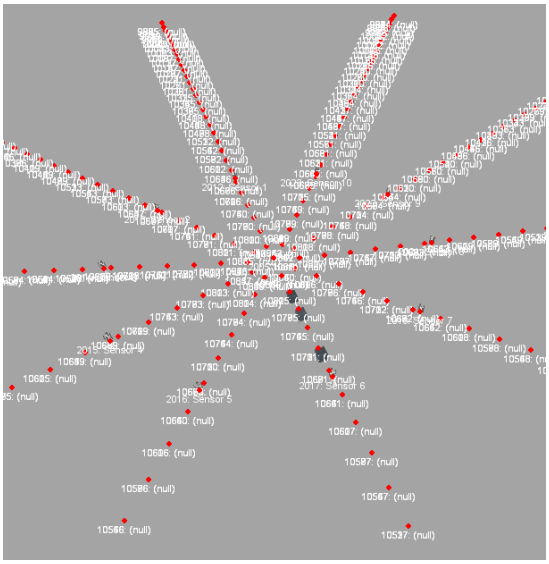

Figure 13: Particle shooting scenario with minimal graphical demands

In the first experiment we simply wanted to test the demands of fire rate alone (which concerns particle construction and destruction). To do this we vary the fire rate and always keep the lifetime equal to it so there can be at most ten particles in the scene at any point in time (since there are ten sensors). The results are shown in Table 4.

Fire rate (ms)	100	50	10	5	3	2
Particles per second	100	200	1000	2000	3333	5000
Frame rate (fps)	64	64	64	32	21	1

Table 4: Variance of fire rate alone

This alone tells us precious little except that constructing and reclaiming the memory for particles extremely rapidly places a substantial burden on our system. Next we look at the demands for simply having particles in the scene. To do this we use a fixed fire rate and vary the lifetime to allow more particles to exist in the scene at a time. The results are shown in Table 5.

Particle Count	10	100	1000	2000	5000	10000
FPS (1000 p/s)	64	64	21	16	0	0
FPS (200 p/s)	64	64	52	32	8	0
FPS (100 p/s)	64	64	64	32	11	0

Table 5: Frame rate with varying particle counts (rate denotes milliseconds between firing ('p/s' stands for particles per second)

These results were a lot more interesting for a couple reasons. First, the drop in the frame rate was gradual until particles reached their lifetime and were reclaimed (yielding the minimum frame rates shown in Table 5). The gradual drop to a minimum shows that particles, by simply existing in the scene (and not necessarily being visible to the player) place a substantial demand on our simulation. Second, the fixed fire rates seem to factor into the number of particles our scene can hold, showing that the rate of particle construction/destruction indeed places a

measurable burden on the simulation.

What about collisions? When particles pass through surfaces they might be absorbed, reflected, or refracted. Surfaces use annotations such as 'concrete' or 'wood' in their name to dictate how collisions should be handled. This all means additional code is being executed and hence should place an additional burden on our system, but by how much? To test this we wrap the emitter in multiple layers of cargo tagged with the 'wood' annotation (having the property of light absorption) to see how performance drops as the number of collisions are raised. Each cargo size was successively larger, creating layers like a Russian Matryoshka doll to ensure particles encounter each face individually. We keep the fire rate and lifetime static at 10 milliseconds and 1,000 milliseconds respectively, which we have shown yields 21 frames per second without collisions. The results are shown in Table 6.

Collisions per particle	0	1	2	3	4	5
Frame rate (fps)	21	21	21	21	21	21

Table 6: Frame rate with collisions (fixed particle counts and fire rates)

We start with this substantial load on the system because earlier results showed that this would make results more sensitive to any drop in performance. However, despite having five layers of collision meshes (meaning 500 collisions a second) we still did not see any hint of dropping performance. This mirrors results found by another researcher in the group that performed a different load test with vastly higher collision counts [4].

Our initial assumptions were that model rendering would exist independently of particle

physics (the former living on the GPU and the later on the CPU). However, when models were introduced it became obvious that this assumption was wrong - the demands of particles and model rendering stack. Making even light models such as the crane visible during the particle load tests showed a significant drop in frame rate. This means that our scenario not only needs to contend with the trade off of graphical fidelity verses performance, but also how much resources will be dedicated to particle physics.

The stacking demands of graphical rendering and particle emulation suggest that it would be advantageous to separate them if possible. Torque was designed for network play, and is hence designed to transmit small deltas between multiple clients running the game. We could take advantage of this by having two simulations (clients) running in parallel. The first would be the front end, using its resources to render the scene and provide control to the user (i.e., player). The second would be headless (not making an effort to display the scene) and dedicate its resources entirely to particle emissions. In scenes with especially burdensome particle physics the back end might be divided further, using a separate system for each emitter (and its relevant particle streams). The problem of keeping the clients in sync is one the first person shooter engine is accustomed to, the front reporting changes made by the player and the back notifying of triggered sensors.

If the scene is purely deterministic (i.e., nothing random and no interactions made by the player) then it would also be possible to pre-process the particle physics. This would mean multiple runs. The first would include the particle streams and produce a log of which sensors fire and when. Future runs would omit the particle simulation and simply use these logs to

indicate when sensors are tripped. This would mean losing the interactiveness of our scene, but in some situations this may be acceptable.

5.3 ADVANTAGES / DISADVANTAGES OF GAME ENGINES

The use of game engines has opened doors in terms of rapid development, making it feasible to provide aspects of the simulation we could not do realistically if coding from scratch. In particular the engine conveys several benefits:

1. Avoids Reinventing the Wheel

   Implementing a simulator with a similar decent degree of fidelity from scratch is extremely time consuming and difficult. Using a preexisting engine allows us to avoid boilerplate development and focus on features of our particular scenario (see related works for examples of other approaches).

2. Obeys Basic Physical Constraints

   First person shooter engines in particular have already been designed to enforce many physical laws we need, such as gravity, velocities, friction, and a mature spatial sense (collisions).

3. Highly Efficient Rendering

   Game engines use a great deal of hacks and abstractions to perform fluently. This is both a blessing and a curse in that care needs to be taken that the engine does not over zealously abstract away features the simulation needs. For instance objects that move

judge collisions based on bounding boxes rather than the more finely detailed collision mesh. This means these objects frequently have difficulty judging how close they are allowed to get to each other. However, this is the cost of fast rendering and the abstractions trend to aid, rather than impair our simulation.

4. Real Time Visuals

   Users and developers are able to see the simulation run in real time, providing insight as to how results are derived and confirmation that it is behaving as desired. This white box model is vital for sanity checking and very helpful in making the simulation's conclusions convincing.

5. Scriptable

   The vast majority of modern game engines are scriptable to help ease development. This allows us to work at a higher level of abstraction, providing instructions for how the environment operates rather than needing to worry about low-level implementation details. In our case this is done via the proprietary TorqueScript, but most engines use the more widespread, open source Lua language.

These attributes allow us to quickly develop simulations that decently reflect real-world characteristics. The fact that we do not need to develop the basic engine itself is particularly important since this constitutes thousands of man hours and would otherwise make rapid

prototyping and deployment impossible. This said, the use of game engines also have several substantial drawbacks too:

1. Limited Fidelity

   Commercial games are made over years by teams of hundreds, if not thousands of professionals with various specialties (developers, modelers, texture artists, animators, audio engineers, etc.). In our work we had three contributors, each an amateur to game development and taking on several roles. This meant the simulation's scope (particularly in terms of art and animation) had to be limited. As a result we omitted things like crane movement and spinning tires, giving the world a rigid feel.

   Fortunately basic game development is not terribly difficult and the vast throngs that work on commercial games are mostly necessary to contend with the diminishing returns of effort in the pursuit of realism. Even if we could replace symbolic representations with more lifelike portrayals it would not be a terribly worthwhile goal. Besides demanding system resources and being pointless to the results of the simulation, the closer we get to a lifelike simulation the more we would contend with the uncanny valley- the natural tendency of individuals to reject imperfect lifelike representations [10]. In this sense the amateurishness of the visuals actually help in making the simulation more acceptable to users. This also helps to accent parts of the simulation we are interested in, namely the movement and detection of cargo through complicated terrain.

2. High Abstraction

The project presents a dichotomy of objectives between realism and real time responsiveness. The game engine lends its support to fluent game play, providing abstractions of the world that can be quickly rendered. Unfortunately this is often inappropriate for our purposes. For instance according to the engine, water is simply a thin sheet animated to ripple. Since this means our boat cannot 'float' in it, we had to flatten the ground under the water and have our boat hover at a fixed distance above it. This sort of hack is just a single example of the work that constitutes game development: learning to work within the engine's representation of the world to trick it to do what we need.

Abstraction also impacts event triggering and notifications. The engine is a best effort service with no hard guarantees for when or how events will be processed. For an analogy, when asked "If a tree falls in a forest and no one is around to hear it, does it make a sound?" game engines reply “no” since the forest is void of the player and hence making a sound would be a waste of resources.

The real time aspect is not strictly necessary for experimental results. However, including it provides a blend of simulation and data visualization that opens additional use cases and makes the work of interest to a wider audience of users. For instance an accurate real

time simulation could possibly be useful to port workers for 'seeing' security holes introduced after sensors are disabled in an accidental collision or during adverse weather conditions.

3. Repurposing

   As mentioned earlier, game engines have a highly abstracted representation of the world, allowing for quick and fluent rendering of the aspects they were designed for. In the case of first person shooters this includes the basic physics for individuals engaged in gunfights and moving about the terrain. Unfortunately this has little to do with the needs of our simulation. This frequently meant fighting against the engine to provide the basic functionality we desired. For instance cargo is commonly treated as a 'StaticShape', moved via matrix multiplication of the desired affine transformation. However, for efficiency's sake the engine ignores collisions on translations, meaning that a piece of cargo cannot tell when it hits the ground if dropped. The solution is to replace the StaticShape cargo with a member of the Item class that looks the same but can be affected by gravity. This is neither elegant nor obvious, but a necessary workaround to get the engine to behave as we would expect the real world to.

4. Poor Design and Incomplete Functionality

   The game industry is not known for turning out good code. It is an extremely highly competitive field where schedules and flashy forward facing functionality are king. Back

end testing, proper API implementation, documentation, and other aspects of 'good programming' are often the first casualties when deadlines loom near. Game engines are no exception – even a developer oriented engine like Torque contains numerous bugs, poorly executed hacks, and even blatantly missing functionality (no-ops).

Documentation is no better. Development often requires guess-and-check to figure out each function's effects. This is partly an aspect of this particular community – Garage Games, the maker of Torque view documentation as a luxury to be sold to developers. They make money selling books and regulating access to the *community* forums (between poor forum navigation and inappropriate access restrictions they are nearly unusable). There is precious little internal documentation and what general references are available are generally hidden in archives on third party sites. The engine has an IRC channel but this seems to be a secret, as the site has erroneous information concerning its location, requiring a bit of scavenging in the archives to figure out where it is available.

5. Antiquated Engine

   Unlike the development philosophy of libraries and utilities, the engines used by games are generally throwaway code, developed for a particular release then rarely improved. This is why documentation and unit testing mean little to the industry – upkeep is not a concern. In our case the Torque game engine was developed for the 2001 release of Tribes 2. As far as the game industry is concerned this makes it fairly old (over a generation

behind – around the age of the PS2). This is part of the reason for the substantial loss in fidelity mentioned earlier concerning the export of DTS models. The engine at best uses eight year old techniques for ray tracing, rendering, and particle physics, which helps to explain the performance issues we have been encountering.

5.4 SUMMARY

In this section we have discussed the issue of scalability in terms of computation and development. Both have introduced concerns for the feasibility of producing large scale simulations, particularly if we wish to maintain the main benefits of using game engines: real time visuals and ease of development. While these are serious issues, they are not insurmountable as we will discuss in the conclusion.

CHAPTER SIX

RELATED WORKS

This section discusses previous works in the fields of simulated trade-off analysis and serious game development (the adaptation of games for pragmatic purposes). Our work is by no means the only attempt to repurpose game engines to relate to the real world, nor the only attempt to even find automated means of analyzing port environments.

Among the first papers we looked at was *A Simulation-Based Approach to Trade-Off Analysis of Port Security* [16]. This looked at the trade-offs between various stakeholder concerns (such as security and throughput) using the response surface method (RSM) to provide a set of Pareto optimal solutions. Analysis was mostly concerned with a simulated supply chain under a variety of loads. This was of great interest since it illustrated concerns our simulation should take into account (interests of port authorities and the need to investigate varying loads) as well as illustrate an off-line means of balancing trade-offs at a high level (i.e. input to their mathematical analysis being generic attributes rather than the use of actual physical layouts as our project does).

Next we looked at another application of game engines in *L3DGEWorld 2.3 Input & Output Specifications* [13]. This paper concerns the technical details of L3DGEWorld 2.3, a data visualization tool utilizing OpenArena (a derivative of the Quake III Arena game engine). The tool consists of four components allowing a measure of flexibility in how it is applied. These include an input daemon, the L3DGEWorld client & server, and an output daemon for translating

in-game actions to the appropriate commands. The paper presents a sample scenario where network administrators detected and blocked a port scan on a greynet via the mechanics provided in the FPS engine. In this case pyramids represented IP ranges, spinning indicated network activity, and being shot caused a block. The tool operates by metaphors which are occasionally somewhat labored. Still, this is an interesting and novel approach to an alternative means of administration. While the environments differ considerably, both have a similar goal in that they are trying to allow users to sift through sizable amounts of real-world data via a player. One lesson we might take from this project is to abandon realism at times in favor usability (such as providing a HUD or aerial view upon request).

The third paper was another application of serious games. *A Simulation Learning Approach to Training First Responders for Radiological Emergencies* [15] discusses the advantages of using game engines to train first responders in the case of radiological emergencies. Specifically, this is concerned with using parallel simulations to provide realistic input to the game engine for things like radiation propagation. The material was at a fairly high level and unfortunately concrete work did not appear to be present.

The next paper was mostly concerned with future research in integrating machine learning to anticipate threats and apply strategic planning. *Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL* [17] provided an interesting look at where our work might go. The state of the art in RTS (real time strategy) game AIs are at heart simple reflex agents. Even most avenues of research are toward pre-learned or map-based tactics due to limited resources and game complexity. This paper provides a different approach using a combination of

case-based and reinforcement learning to allow AIs to grow from experience and translate those experiences to different environments.

The proposed design delegated the scope of decision making to have a multi-tiered approach where each layer's output are the goals for the next with temporal progression (i.e. lower layers run more frequently). This allows for occasional strategic thought at the top and fast tactical decision making at the lowest levels. While this could easily be applicable to infiltrators for our port scenario (i.e. combatants trained in urban environments trying to take control of the port), it offers limited usefulness to cargo inspection. The issue is scope - sneaking a given piece of cargo past static sensors is a matter of finding gaps, not high level strategic planning. However, the port simulation is planned to simply be a component in a larger scheme which does include strategic planning and could likely benefit from transfer learning.

Another paper related to our work is *Situated Design of Virtual Worlds Using Rational Agents* [9]. Conventionally game environments are crafted by developers by blending rendered 3D models and scripted events to provide the desired scenes. Though there has been a few commercial applications of auto-generated environments (for instance dungeons in Diablo II) this has mostly stayed in the 2D realm to cut down on complexity. This paper discusses the use of situated rational agents (AIs with a spacial sense) to craft 2D and 3D environments using a predefined design grammar (viable geometric transformations like rotation and mirroring) and descriptions of function (tags for their attributes, purposes, and associated scripts) to automatically structure scenes according to given constraints and objectives.

The port simulation is at its heart an analysis and optimization problem. Concepts

proposed in Maher's paper are not directly related to our work (since we are reflecting an established sea port, not making one of our own). However, for optimization Maher's approach has the added benefit in that if the reasoning was provided to a player agent (i.e. someone that needs to obey physical laws like gravity and collisions), then we can draw added information from the process of optimizing the scene. For instance, if we are trying to rearrange cargo to optimize sensor coverage, then a simple optimization algorithm might make several decisions that, though technically correct, are completely unhelpful as a real world solution. For instance its tactics might include...

... placing all cargo on a distant mountain.
... placing all cargo under the ground or suspended high above the scene.
... stacking all cargo into a single, high tower.

Though we could take an iterative approach where we script in rules disallowing these sorts of solutions, forcing the AI to use a player's body to transport the cargo would eliminate most of these problems. For instance players cannot climb mountains, go underground, or fly up to place the cargo in impossible locations. In addition, since the player agent is constrained to specific movement speeds and activities, we could log what it does to optimize the arrangement so it can be used later to exemplify how the rearrangement can be done in the real world and to provide an estimate for its viability. With a weighting of real-time and price to game time activities, estimating the time and cost of such changes would be trivial. In addition physical constraints

force the agent to take realistic approaches. For instance swapping two pieces of cargo would require that the agent have a temporary location for the exchange which might radically change a solution's viability.

Finally we concluded with two papers focused on the rendering of large environments, focusing on trade-offs between fidelity verses performance. The first of these was *Visual Attention Models for Producing High Fidelity Graphics Efficiently* [3]. It proposed using high-level task maps and low-level saliency maps to anticipate where users will be looking in the scene to render those parts to a higher level of detail. This takes advantage of weaknesses in human vision, specifically that our foveal focus (a tiny region of vision with the best resolution through a dense packing of cones) dances about the scene to make a detailed inspection of only a few relevant objects. If the foveal regions can be successfully predicted the resolution of the rest of the scene can be dropped without being noticed by the user.

There are two methods for tracking what will draw attention in the scene. The first is saliency maps which use visual psychology research to track the evolutionary tendencies for our eye to be drawn to specific features of the scene. These include edges, abrupt changes in color, sudden movements, high degrees of contrast (such as a candle in a dark room), and the most expressive areas of the face like the eyes and lips. The second method uses task maps, the draw of the user's attention when given a specific task like counting the balloons in a scene.

Things that lie outside the viewer's area of attention incur 'inattentional blindness', meaning that details and often even their presence or absence go unnoticed. In their experiment subjects were tasked with counting pencils in the scene, then asked if they noticed a drop in

visual quality. The results were that 75% of the subjects noticed the drop if anti-aliasing was stripped from the whole scene while only 5% noticed if the foveal region around the pencils retained a high quality. When the viewer's focus was unpredictable it is not as easy to make the low resolution go unnoticed. When the viewer's attention shifts the new foveal region must have full resolution within five milliseconds of fixation or the observer will detect the low resolution.

This paper gives fascinating hints as to how partial rendering should be applied to improve performance with the least perceivable impact to users. Since users of our simulation are concerned with the transport of contraband, a task map would suggest anything outside the immediate route that cargo will take could possibly suffer a substantial drop in fidelity without being noticed by the user. Saliency maps would also suggest things like the orange cargo and visible particle streams will draw the most focus. Usability studies with eye tracking would be needed to confirm which details could be safely dropped.

The second paper, *Rendering of Large and Complex Urban Environments for Real Time Heritage Reconstructions* [19], strives to find low-level optimizations to allow for greater rendering of large, high-fidelity urban environments. Some of what the paper discusses are common graphical techniques, customized to provide added performance in their simulation. This starts with the scene graph, the representation of objects contained within the environment. Their testbed uses a tree structure to limit the memory consumption of duplicate objects (in our scene things like cargo) and contains annotations for culling and detail level optimizations discussed later.

Next this moves on to the view frustum, which culls objects for which the bounding box

lies outside the player's field of vision. The vision is treated as a cut pyramid, not only having side bounds but a limited reach. View frustum are a conventional part of the graphics pipeline and handled for us automatically by Torque. However, the discussion brings up a simple means for improving performance: limiting the player's field of vision and hence culling additional objects from the scene. The change would be trivial to implement (changing a single parameter in Torque) and would mean giving the scene a foggy appearance with only silhouettes of distant objects being visible. The paper also discusses occluder shadows, the culling of objects entirely eclipsed by things like buildings.

The paper goes on to discuss the limiting of vertex counts. Their testbed, in full fidelity, comprised of 321,602 polygons which is both several times larger than our port and beyond what is feasibly rendered in real time on current workstations. To address this they propose the application of levels of detail using height maps and Real-Time Optionally Adaptive Meshes (ROAM). The former is an improvement already proposed in our work and the later consists of a substantial low-level improvement to on-demand object rendering.

Throughout the paper all improvements were made via changes in OpenGL. Torque abstracts away this level of control for our simulation but the results derived in this paper seem promising enough that it might be worth exploring these sort of low level improvements in the future.

CHAPTER SEVEN

CONCLUSIONS

This section discusses the conclusions and lessons learned from our research. Though the game engine provided a framework in which to work, development went surprisingly slow. Both sculpting and scripting demanded a great deal of time - the former due to the high degree of detail needed to avoid distracting blemishes and the later because of issues with the underlying engine which was largely designed for different needs. Conventional game development is replete with hacks and hard coding, which are viable for simple terrains but quickly becomes unmanageable as the scenario's complexity grows.

In the second section we broached a question: how effective are the game engines (particularly Torque) for simulation purposes? This question has no easy answer since development had its ups and downs. Community support for DTS exporting for instance was excellent while engine documentation was lackluster to say the least.

Though many of the issues we raised are specific to Torque, we would not have necessarily been better off with another engine. Poor coding is a symptom of the game industry, which is characterized by fast deadlines and a low need for upkeep. As game development moves more toward user generated content we may see improvements on this front. Lua and the trend toward standardization and scripting are good indicators that the industry is moving in the right direction, toward development practices that address many of our issues (better coding, stability, and documentation).

Despite the drawbacks raised, however, our simulation demonstrates a working level of detail that would not have been possible if not for having an established environment to work in. In addition, as game engines and hardware advance both difficulties in using the engine and issues concerning performance will likely become a thing of the past.

As for the question of feasibility, we have shown that it is possible for a tiny group of developers with no prior game building experience to put together a basic simulation in a relatively short period of time. Though we have not reached the point of answering the interesting research questions like how well this reflects the real world, we have provided a testbed environment where these lines of research can take place.

Finally, as to the question of scale our experimental results show an inability to support large scenes with a decent level of graphics. Even a dozen moderately detailed models such as the ship and crane cannot exist in the scene without severely impacting the performance on decently high end workstations. With our current engine, hardware, and detail level it would be a struggle to simulate a fraction of a port, let alone its full operations.

Further modifications could both help and hurt this, for instance our simulation had the simplest kind of lighting possible – bright ambient illumination. The introduction of localized light sources will greatly increase demands on the graphics card. However, future development could also introduce varying detail levels (and attribute supported by DTS models) to simplify rendering of distant shapes.

CHAPTER EIGHT

FUTURE WORK

Our work was largely toward the development of a functional testbed environment, and development time precluded the pursuit of more general research questions regarding simulation of a port locale. This section discusses likely improvements and directions of later discovery. Potential research falls into two broad categories: continuing investigation into using game engines for simulations and instance based investigations.

The former concerns extensions to our testbed and addressing the shortcomings we have encountered. Most importantly this includes addressing the performance issues that keep us from having sizable simulations. Among commercial games using the Torque game engine (such as Age of Time and Civil Disturbance [6]) the use of curved faces is avoided and a large effort put toward keeping the vertex count down (either through sparse environments or low model detail). In some cases the limited fidelity is surprisingly subtle, cleverly using textures and object placement to cover the actual simplicity of the scenes. We have put some effort toward simplifying the models we use and future efforts should continue doing so. Ideally this would be taken over by modeling artists, who are better aware of tricks to achieve both low detail and decent appearances.

A relatively easy means of improving performance would be to make use of the “detail level” functionality provided by DTS models. This allows models to be replaced by a simpler representations (with a lower vertex count) when the player is too far away to make out specific

details. By using basic geometric shapes to roughly approximate the silhouette we can reduce the graphical needs of the port to a fraction of what they are now, possibly allowing us to accommodate a dozen times more objects. The variable detail models exist independently of the collision mesh, so regardless of how an object is being rendered, we can continue to respect particle collisions at any level of fidelity we wish. Ideally the level of detail would be controlled via procedural means (automated subdivisions from a template), allowing us to programmatically control the vertex counts in our simulation. Finally, usability studies could deem a high degree of visual fidelity to be unnecessary for making our simulation intuitively understandable, allowing us to drop efforts toward realism entirely.

Simulating particles can take a substantial amount of resources. Since this has been shown to stack with the burden of rendering it would be advantageous to keep the client from trying to do both at once. Two possible solutions would be parallelizing the simulation (tracking particles on separate clients) and pre-processing particle physics so subsequent runs could use the generated results.

Another issue to address is the rigid, unreal feel our simulation currently has. Some solutions would be the inclusion of animations (obvious candidates being wheels and cranes), and the inclusion of point light sources (such as lamps). Both of these would conflict with the previous goal of improving performance and the tradeoffs should be carefully considered. Finally, the most noticeable deficiency is the spartan feel of the environment. Buildings, roads, and other basic features would go a long way toward giving the port a more complete feel.

Finally, experimental benchmarks for how improvements to the CPU and/or GPU affect

performance might give insight to the rate at which the scenario can be expanded as hardware improves. Are we able to fully take advantage of Moore's law or does the engine impose some limitations? Potential experiments could use virtual machines with support for direct rendering to provide sandboxed environments and regulate the resources available. In the case of GPUs it might be important to check chipsets provided by both of the major manufacturers (Nvidia and ATI).

The other branch of research addresses the fundamental question that was the reason for making these simulations at all: how faithful can it be for reflecting the real world? Answering this question is vital to proving the usefulness of game engine based simulations and will require the detailed modeling of real locations, complete with experimental results that can be validated with their real world counterparts. Once this approach has been validated for the port environment, the crossover benefits for other environments (like airports and highways) would pose an interesting point for future research.

BIBLIOGRAPHY

1. "About." Lua. 11 Feb 2009. PUC-Rio. 15 Mar 2009 <http://www.lua.org/about.html>

2. "blender.org." Blender. Blender. 15 Mar 2009 <http://www.blender.org/>

3. Chalmers, Alan, Kirsten Cater, David Maflioli. "Visual Attention Models for Producing High Fidelity Graphics Efficiently." *Proceedings of the 19th spring conference on Computer graphics* 19(2003): 39 - 45.

4. Christiansen, Allen. "Modeling Sensors and Threats in a Three-Dimensional Real-Time Simulation of a Seaport Environment."

5. "FRAPS." FRAPS. Beepa Pty. 15 Mar 2009 <http://www.fraps.com/>

6. "GarageGames." Game Development Tools and Software. Garage Games. 15 Mar 2009 <http://www.garagegames.com/>

7. Johnson, Damian, Allen Christiansen, Lawrence Holder. "Game-Based Simulation for the Evaluation of Threat Detection in a Seaport Environment." *Entertainment Computing* -

ICEC 7(2008): 221-224.

8. Kaniuth, Roland. "GNSS Software Simulation." Institute of Geodesy and Navigation. Institute of Geodesy and Navigation. 25 Mar 2009 <http://ifen1.bauv.unibw-muenchen.de/research/gnss_simulator.htm>.

9. Maher, Mary. "Situated design of virtual worlds using rational agents." Proceedings of the second international conference on Entertainment computing 38(2003): 1 - 9.

10. Mori, Masahiro. "The Uncanny Valley." Energy 7(1970): 33-35.

11. Morkel, Chantelle, Shaun Bangay. "Procedural Modeling Facilities for Hierarchical Object Generation." *International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa* 4(2006): 145 - 154.

12. Müller, Pascal, Peter Wonka, Simon Haegler, Andreas Ulmer, Luc Van Gool. "Procedural Modeling of Buildings." *International Conference on Computer Graphics and Interactive Techniques* (2006): 614 - 623. Print.

13. Parry, Lucas. "L3DGEWorld 2.3 Input & Output Specifications." 22 Feb 2008 25 Mar 2009 <http://caia.swin.edu.au/reports/080222C/CAIA-TR-080222C.pdf>.

14. "Port of Seattle." Wikipedia. 25 Mar 2009 <http://en.wikipedia.org/wiki/Port_of_Seattle>.

15. Sanders, R. L.. "A simulation learning approach to training first responders for radiological emergencies." Summer Computer Simulation Conference (2007):

16. Sekine, Junko. "A simulation-based approach to trade-off analysis of port security." Winter Simulation Conference 38(2006): 521 - 528.

17. Sharma, M.. "Transfer learning in real-time strategy games using hybrid cbr/rl." Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (2007)

18. "The Entertainment Technology Curriculum." The Entertainment Technology Center. Carnegie Mellon University. 15 Mar 2009 <http://www.etc.cmu.edu/curriculum/index.html>

19. Willmott, J., L. Wright, D. Arnold, A. Day. "Rendering of Large and Complex Urban Environments for Real Time Heritage Reconstructions." *Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage* (2001): 111 - 120.

20. Wimmer, Michael, Peter Wonka. "Rendering Time Estimation for Real-Time Rendering." *Proceedings of the 14th Eurographics workshop on Rendering* 14(2003): 118 - 129.