A High Performance Framework for Coupled Urban Microclimate Models

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Matthew C. Overby

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

Dr. Peter Willemsen

November 2014

Acknowledgements

In 2010 Dr. Pete Willemsen offered me a position as an undergraduate research assistant to work on an application that modeled radiation in urban environments. This escalated into graduate research, giving me the opportunity to experience more than I ever thought possible. I owe my success to Dr. Willemsen, who could not have been a better advisor and friend.

I've had the pleasure of meeting some of the most intelligent, motivated, and friendly people as a member of the GEnUSiS team. Scot Halverson, who guided me into the project as a newbie computer scientist. Dr. Eric Pardyjak, who helped me learn and achieve so much. Brian Bailey, who spent countless hours helping me understand the physics of heat transfer modeling, and put up with my occasional outbursts of frustration. Dan Alexander, Kevin Briggs, and Rob Stoll, thank you for the same.

A special thanks to Lori Lucia, Clare Ford, and Jim Luttinen, who have been extremely helpful over the last few years. I'd also like to thank Dr. Richard Maclin and Dr. Marshall Hampton for their role on my graduate committee.

I am grateful for everyone's help and inspiration.

Dedication

To my incredible parents. Your compassion, support, and guidance throughout these years has given me the courage and will to achieve my goals. I love you both so very much.

Abstract

Urban form modifies the microclimate and may trap in heat and pollutants. This causes a rise of energy demands to heat and cool building interiors. Mitigating these effects is a growing concern due to the increasing urbanization of major cities. Researchers, urban planners, and city architects rely on sophisticated simulations to investigate how to reduce building and air temperatures. However, the complex interactions between urban form and the microclimate are not well understood. Many factors shape the microclimate, such as solar radiation, atmospheric convection, longwave interaction between nearby buildings, and more. As science evolves, new models are developed and existing ones are improved. More accurate and sophisticated models often impose higher computational overhead.

This paper introduces **QUIC EnvSim** (QES), a scalable, high performance framework for coupled urban microclimate models. QES allows researchers to develop and modify such models, in which tools are provided to facilitate input/output communications, model interaction, and the utilization of computational resources for efficient simulations. Common functionality of urban microclimate modeling is optimally handled by the system. By employing Graphics Processing Units (GPUs), simulations within QES can be substantially accelerated. Models for computing view factors, surface temperatures, and radiative exchange between urban materials and vegetation have been implemented and coupled into larger, more sophisticated simulations.

These models can be applied to complex domains such as large forests and dense cities. Visualizations, statistics, and analysis tools provide a detailed view of experimental results. Performance increases with additional GPUs and hardware availability. Several diverse examples have been implemented to provide details on utilizing the features of QES for a wide range of applications.

Contents

List of Figures

1 Introduction

The urban microclimate is a governing factor of building energy demands and human living conditions [36]. The Urban Heat Island (UHI) is a phenomenon in which dense urban layouts trap heat, resulting in a higher ambient temperature than surrounding areas [5]. With increasing urbanization of major cities, the mitigation of UHI is a growing concern [13]. As the surrounding air temperature rises, buildings require more energy to cool their interior. The additional energy usage may exhaust more heat into the environment, propagating the buildup of heat. But, the process by which these interactions occur is complex. Researchers, urban planners, and city architects can benefit from sophisticated simulations of urban form to investigate how to reduce building and air temperatures, and design more environment friendly city infrastructure. This may manifest as ideal building shapes and sizes that have varying effects on the microclimate. For example, air currents can transport heat out of the urban layout [5]. Green rooftops can be used to absorb radiant heat from the sun [13] and insulate buildings during the winter. Specialized urban materials can be used to reflect radiation back into the atmosphere. Many different strategies can be explored with the help of computer simulations.

1.1 Scientific Modeling

The complex interactions between urban form and the environment are not easily computed. Many factors shape these interactions. To quantify and describe these

effects, researchers and scientists rely on **modeling** the physical processes to better understand them. This typically involves composing observed phenomenon into a physical and mathematical representation. These representations are a cornerstone of many scientific fields and are further used to help researchers gain an understanding for the object or system they illustrate. The **coupling** of models has multiple meanings. Often it describes the use of multiple models for the same task to provide redundancy, which has been shown to improve accuracy in model prediction [14]. Alternatively, it can be used to describe using different models that provide input and output to one another, composing a more diverse and sophisticated simulation.

Typically, models are validated by comparing their output to empirical observations recorded in the real world. A model's strength is its ability to accurately predict the outcome or structure of something given the correct initial state or input. However, models are rarely without error. Because of the complexities of the real world, models can only approximate and are not complete representations. In the case of climate modeling, research has shown that different models have varying degree of accuracy depending on application [14]. This error often stems from the user's ability to supply the model with a correct initial state, or applying the model to conditions it was not validated against or designed for [1]. Despite this, models remain an integral part of microclimate analysis. As science progresses these errors are reduced by *developing new models and improving existing ones.*

A common goal of urban microclimate modeling is to correctly simulate the surface and air temperatures within the urban domain [46]. In this application, researchers can better understand how certain structures, materials, and layouts impact urban heat. This is often done by modeling the many elements that affect urban heat, such as solar irradiation, longwave exchange, and heat exchange due to air currents. But the complexity lies in the components that govern these elements. For example,

solar irradiation at a given point on earth depends on the sun's altitude, which will vary with time of the year and geographic coordinates. In addition, this radiation can specularly and diffusely reflect based on surface reflectance models, atmospheric particulate, and participating media. The result of these dependencies are that if any one of these individual components does not reliably represent the physical world it is modeling, the whole may lead to misleading or incorrect conclusions. Thus, assumptions such as material properties or boundary conditions must be made to simplify the model. *Because of this, models are often designed for specific needs.* Inputs and outputs are tuned to answer certain questions. Few models can answer all questions.

1.2 Computability of Microclimate Models

The computational cost of microclimate modeling is another major challenge posed by the complexities of the real world. It is not always feasible, or required, to simulate every physical interaction. Many urban climate modeling publications ([9, 24, 21]) do not report much (if any) detail on simulation run time or hardware portability, and instead focus on model accuracies.

The research in this paper takes advantage of advanced computational methods to accelerate scientific models. One such method is the use of Graphics Processing Units (GPUs) for parallelizable functions. A GPU is a powerful parallel processing device, consisting of hundreds to thousands of cores designed for high throughput. Often they are used to render computer graphics intended for display. Driven by the video game industry, GPUs have seen tremendous growth in capability. With the introduction of general purpose GPU computing libraries such as NVIDIA Compute Unified Device Architecture (CUDA), developers and researchers are able to use this

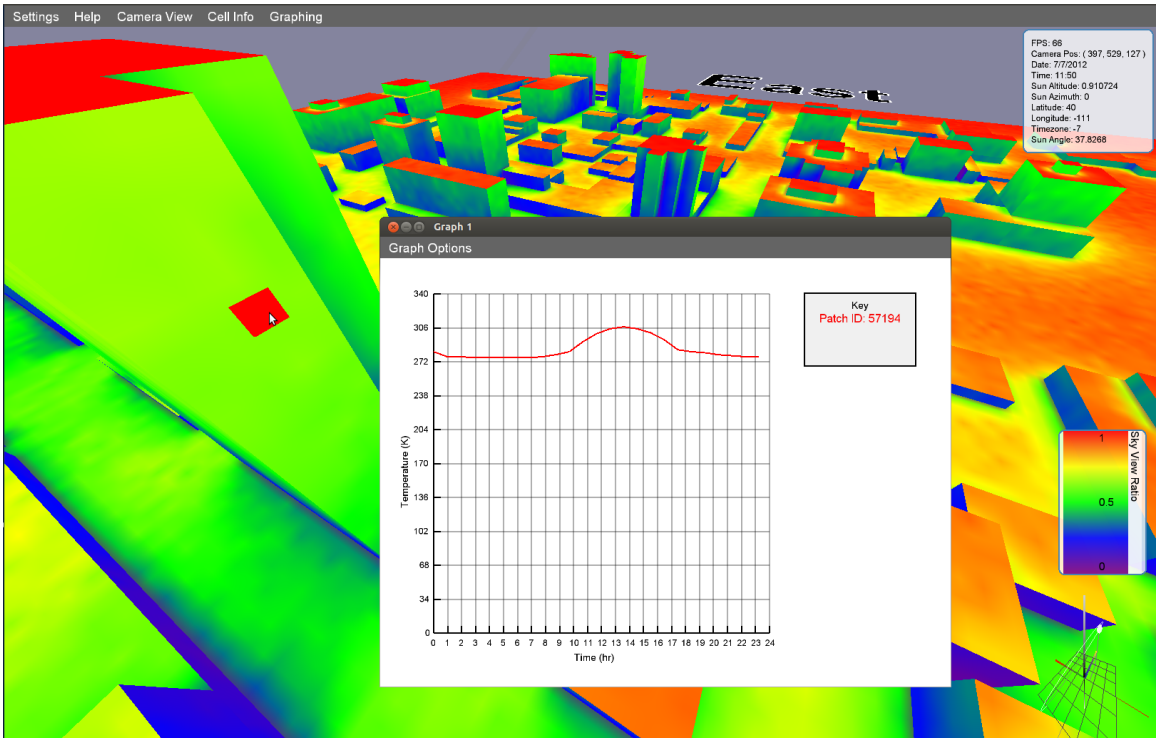


Figure 1.1: The interactive graphical user interface of QUIC EnvSim. Users can simulate and visualize components of the environment such as sky view factors and surface temperatures (shown).

hardware for other computational tasks [32]. More information on GPU computing is given in section 2.4.

1.3 QUIC EnvSim

To facilitate the coupling of models and enable acceleration with GPUs, this paper presents a dynamic, high performance application programming interface (API) for microclimate modeling, called QUIC EnvSim (QES). The overall goal is to provide a framework to enable the development of coupled models, as well as provide a flexible interface for tuning input and output to answer a wide range of research questions. Specifically, this research has three primary objectives:

1. Design a dynamic, extensible framework for urban microclimate modeling.

2. Develop and test system components to facilitate high performance computing techniques. To automate certain functionality and allow future researchers to utilize such techniques.

3. Implement several important models for urban microclimate, including radiation transport, surface view factor, and a land surface model. Illustrate how models can be coupled, ablated, and validated.

As new models, paradigms, and techniques are developed, they can be introduced into the QES framework with ease. The tools, samples, and tutorials presented in this research can aid the creation of new models, in which many of the computational challenges are automatically handled by the software. To accelerate such computations, QES takes advantage of GPUs. From a simple desktop workstation to a powerful multi-GPU remote server, QES attempts to automatically scale the workload based on hardware availability.

In addition to providing system components, several models of the common influences of the urban microclimate were implemented and validated. These models include sky, sun, and wall viewfactors, solar irradiance and longwave exchange, and a land surface model (LSM). These models are available to be coupled with new model implementations in future developments. The domain can include many building types and materials, vegetation, and virtual sensors to gather information about the environment. QES can load standardized input from environmental data sites such as MesoWest to supply models with input. MesoWest offers decades of observed weather data for hundreds of locations across the United States [18].

The simulation of viewfactors and radiation is done with a technique called ray tracing. Ray tracing is used to sample an environment by tracking the path of a

ray as it interacts with objects in the domain. Historically, Monte Carlo techniques such as ray tracing are associated with a high computational cost [42]. However, the parallelizable nature of ray tracing lends itself to gaining significant acceleration when computed with GPUs. The programmable engine and API that is used for this is NVIDIA OptiX, discussed further in section 2.5. These results and other models are post-processed with CUDA. The many examples and tutorials provide a foundation and illustration to allow new users to implement their microclimate models using the resources of QES.

QUIC stands for Quick Urban and Industrial Complex, a dispersion modeling system developed by Los Alamos National Laboratory [8]. It contains applications to simulate 3D wind flow, pressure fields, and particle dispersion in urban environments. In addition, urban domains can be created with an interactive city builder. QUIC EnvSim was designed to be able to take in the outputs of these simulations as input.

Several models have been implemented in QES to demonstrate the dynamic capabilities of the system. Modular test cases show these models can be applied to complex domains such as dense forests, downtown Salt Lake City, Utah, and the University of Minnesota Duluth. Visualizations, statistics, and analysis tools provide a detailed view of experimental results. Models can be arbitrarily ablated or adjusted without compromising the integrity of the framework. QES can scale work to multiple GPUs and utilize the resources of a powerful multi-GPU server, or limit itself to a MacBook Pro.

1.4 Symbols and Terminology

Documenting research that spans multiple scientific fields has many challenges. One of which is the consistency and definition of various terms and terminology [25].

Because this work spans the disciplines of meteorology, fluid dynamics, and computer science, an effort was placed on defining all terms, abbreviations, and representing symbols in a way that is consistent with literature.

1.4.1 Text Styles

This document will make use of text styles to help illustrate points and key concepts. In general, these styles are:

1. **Bold**: Definitions and important terms.

2. *Italics*: Emphasis to help the reader understand important content or key points. Specific use will vary with context.

3. `Teletype`: Specifically relates to objects, tools, and computational components. Often directly correlates to classes, functions, or variables of the implementation and associated modules.

4. SMALLCAPS: Defines phases of execution within the program pipeline. The four phases, SETUP, INITIALIZATION, SIMULATION, and TERMINATION, are discussed in section 3.1.

1.4.2 Terms

QUIC EnvSim contains a collection of computational modules, designed for interoperation and extension. The modules themselves encapsulate numerous microclimate models, such as computing the position of the sun or simulating vegetative longwave emission. These models act upon validation test cases in which simulations are executed and output examined. A user creates these models and validation test

cases. The following terms that will be used throughout this paper are defined to avoid ambiguities.

- **Model**: A physical and numerical representation of real-world phenomenon. Models have the potential requirement to be coupled, in which one model's input is a different model's output.

- **Module**: A computational implementation of a collection of models, labeled in `teletype`. QES contains several modules, discussed in chapter 3. Because of the potential requirement for coupled models, modules may also inherit this requirement.

- **Tool**: A computational device or system component that facilitates the operations of QES. These are objects that are globally shared, allowing models and modules to communicate through designated channels.

- **Test Case**: A test of a model or collection of models with specific input, settings, and functionality. A test case can include specific set of buildings and trees, whereby certain models act upon this domain. Examination of the output is handled by the test case, such as verifying correct model output or comparing the result of different models.

- **User**: Because this document describes the computational framework of a modeling system, many references are made to a "user" of the system. This could be a programmer that is creating new models or validation test cases, and interacts with the code directly. A user of this type may be a researcher or scientist. Another type of user is one that interacts with the completed system or visualization interface, and is more concerned with simulation output. They will generally have no code-level interaction, such as urban planners and architects.

2 Background

2.1 The Urban Energy Balance

To mitigate the UHI, we must reduce temperatures within the environment. Known methods for this include applying reflective paints to bounce solar energy back into the atmosphere, increasing surface exposure to the sky to promote cooling at night, or adding vegetation in the forms of trees or green rooftops to absorb solar heat and air pollutants [13]. To find an optimal solution, we must simulate these conditions, compare them, and evaluate.

All objects have an **energy budget** that defines the relation of incoming, outgoing, and stored energy. This energy budget is used to calculate surface temperatures, discussed further in section 3.5. We define the energy budget of a surface as followed:

$$0 = Q^* - Q_h - Q_g - Q_e. \tag{2.1}$$

This equation is can also be called the **energy balance equation**. Q terms represent heat fluxes, in which heat is released or absorbed by the medium as a result of energy transfer. Q^* is the **net radiation** that is incident, reflected, and emitted by the surface. This quantity is discussed in more detail in section 2.3. Q_h is the **sensible heat flux**, a direct result of the *exchange of heat* through conduction and convection. For example, when a cool surface is heated by a warmer air temperature, its sensible heat flux will increase. Q_e is **latent heat flux** and is produced during

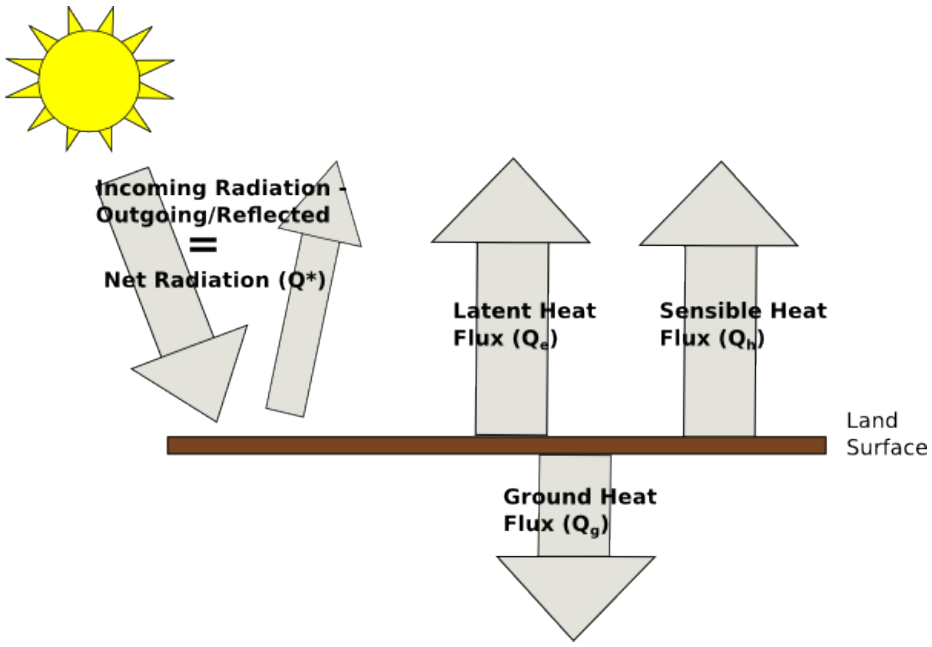


Figure 2.1: The energy budget of a land surface.

a *phase change* with a constant temperature, such as the evaporation of water. The energy required to make this phase change comes from the water. Q_g is **ground heat flux**, also known as **conductive heat flux**, in which energy is transferred to connected or subsurface layers. Each of these terms represent a component that needs to be modeled or simulated. Like all climate models, the complexity and specifics of equation 2.1 may vary with application.

Solving this equation at every surface in the domain enables us to explore the effects of urban form on the environment. The model that encapsulates the calculation of the equation and its inputs is a **land surface model** (LSM). We can investigate the climatic consequences of urban layouts, materials, vegetative infrastructure, and more by evaluating and modeling this equation. The focus of current research and future works will be the development of an energy balance equation for vegetation in urban domains.

There are many components to equation 2.1. Net radiation (Q^*), however, is a focus of this research because of its computational complexities and importance in the surface energy budget. It is necessary to provide a high performance, physically realistic radiation transfer model that can simulate energy in urban domains with vegetation. This will allow future researchers to progress toward defining more detailed and efficient models for the other terms of the surface energy budget.

2.2 Previous Work

QUIC EnvSim represents a continuation of work to model equation 2.1. The first iteration, called QUIC Radiant, was developed by Josh Clark under the advisement of Pete Willemsen and Eric Pardyjak [11]. In this iteration, the GPU accelerated ray tracing engine NVIDIA OptiX was used to compute sun, sky, and wall view factors in simple urban domains. These view factors were then used to determine incoming

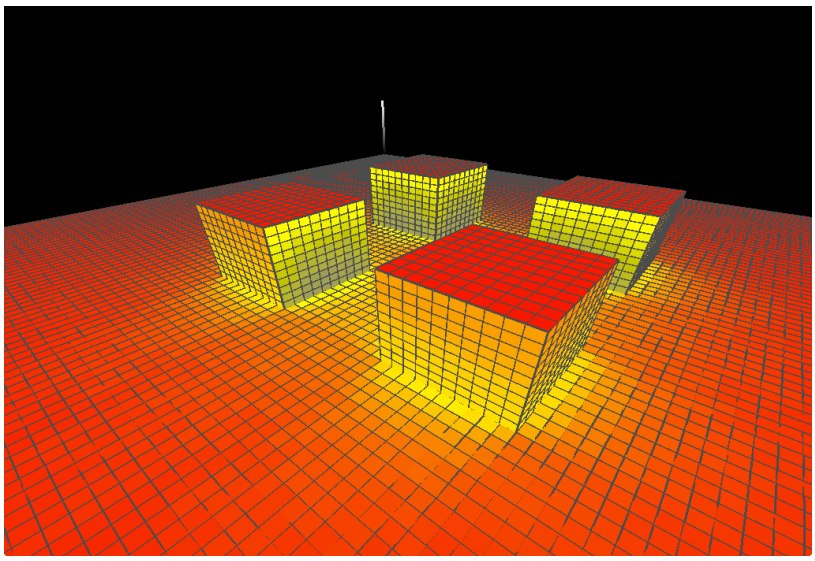


Figure 2.2: Sky view factors of a four-building test case produced by Clark [11]. Used with permission.

and outgoing radiation as terms for an energy balance equation. The software was able to compute the surface temperatures at discretized points within the domains. These view factors and energy balance terms could be interacted with and visualized with OpenGL. Figure 2.2 shows the visualization produced by his work.

The second iteration developed by Scot Halverson improved several aspects of the software. The efficiency of the computations were drastically improved, allowing for larger and more complex domains. Scenes such as downtown Salt Lake City, Utah, covering several kilometers, could be simulated. Visualizations were improved to better convey the intensities of heat and the terms that govern it. Solar energy could be more correctly modeled to include reflections, as well as more sophisticated and physically realistic temperature calculations. Figure 2.3 shows the visualization produced by his efforts. Much of the work involving the visualization of surface properties is still used by the QES framework.

This paper owes a great deal of appreciation to the trailblazers of the past. Clark and Halverson provided a road map of efforts which helped develop a well formed research direction. They contributed a great deal of insight to the complex tasks of urban microclimate modeling. Their guidance made QES possible.

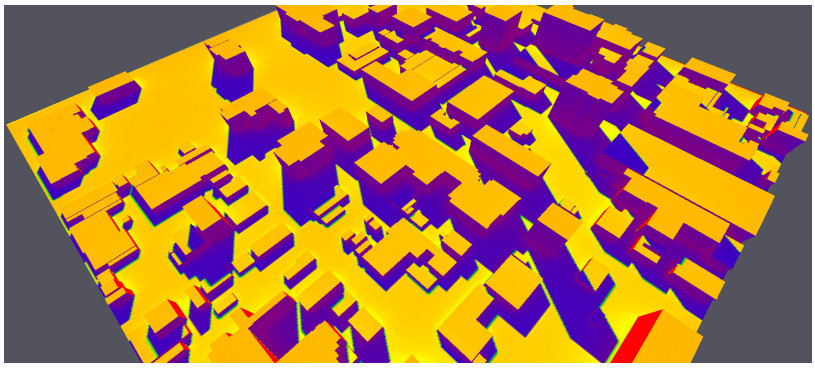


Figure 2.3: Sun view factors of downtown Salt Lake City, Utah, produced by Halverson [15]. Used with permission.

2.3 Radiation in Urban Environments

A key component of the urban microclimate is net radiation, Q^*. This is a driving factor of the surface energy budget during daylight hours. It is a major point of difficulty in microclimate modeling because accurately representing the physics of radiation is computationally expensive. This section will provide detail on some of these challenges.

2.3.1 Electromagnetic Radiation

The term **radiation** encapsulates the event of transmitting energy from one medium to another through waves or streams of particles. It may relate to nuclear radiation, harmful, high frequency *waves* that pass through buildings and material and cause damage to living tissue. It's also used by a certain household kitchen appliance that uses *microwaves* to excite water molecules and heat a burrito. Many things such

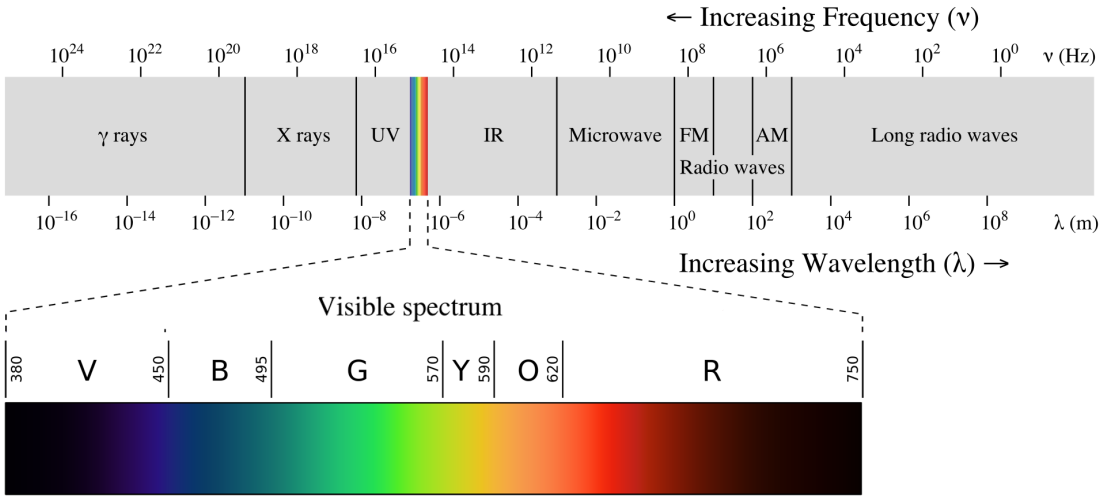


Figure 2.4: The electromagnetic spectrum: associated names for frequency ranges. "EM spectrum" by Philip Ronan is licensed under CC 3.0.

as visible light, radio waves, and X-rays are forms of **electromagnetic radiation** (EMR). It is called electromagnetic, because physically it is both electric and magnetic. We can differentiate between different types of EMR by its wavelength, the distance from one wave peak to the next. Ultraviolet, X-rays, and gamma rays have a relatively short wavelength. Radio, microwaves, and infrared are larger. Visible light has wavelengths somewhere between them. When describing EMR it is convenient to interchange the terms *radiation*, *energy*, and *light*.

Two wavelengths of radiation are often considered when modeling urban climate because of their dominant affect on urban temperature [41]. **Shortwave radiation** (SW) is emitted by the sun. As it travels through the Earth's atmosphere, it is absorbed and scattered by gas molecules and particulate. Eventually, some of that energy will reach the surfaces on Earth and be absorbed, scattered, and reflected to other objects, or back up into the atmosphere. In addition, all things that have a temperature emit **longwave radiation** (LW). Like shortwave radiation, longwave has the potential to be absorbed, scattered, and reflected by the atmosphere and objects on Earth. The specific wavelengths of SW and LW energy vary in literature. In microclimate modeling, it is easiest to define them as a broad spectrum, since nearly all simulated radiation will be either SW or LW.

Vegetation is more efficient at absorbing specific wavelengths of SW radiation for photosynthesis. Thus it is often convenient to consider solar energy in the forms of **photosynthetically active radiation** (PAR) and **near-infrared radiation** (NIR). As its name suggests, plants are more adept at absorbing the former [19].

An important characteristic of the wavelength is the temperature associated with it. Radiation with a shorter wavelength is able to carry more particles over the same period of time. The smaller the wavelength, the higher the temperature. This remains true for all EMR. We can use this information to find how much heat is generated on

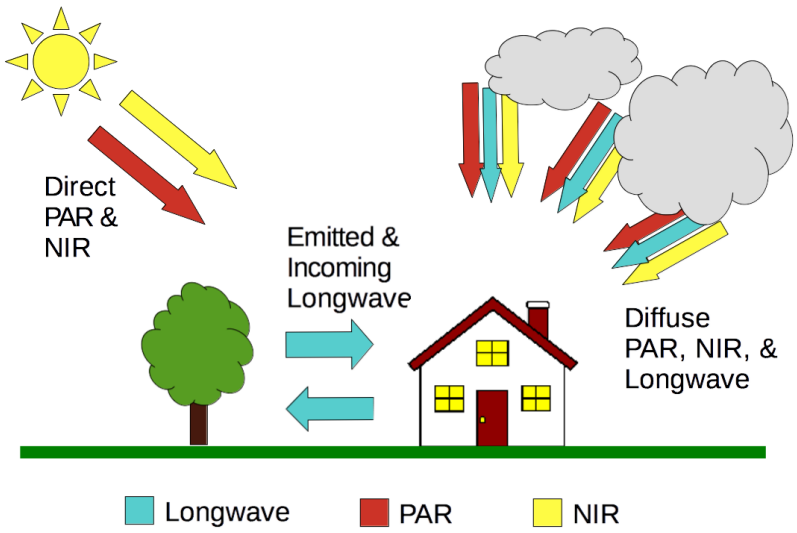


Figure 2.5: Exchanges of radiation for LW, PAR, and NIR in an urban domain.

surfaces from sources such as the sun.

There are different units that are convenient when describing radiative intensities. Often energy is calculated with respect to time (joules per second) in form of watts (W). However, some models take the area of emittance or irradiance into account, resulting in watts per meter squared (Wm^{-2}). The latter is often referred to as radiative **flux**.

2.3.2 Physical Properties of Urban Materials

Many different physical objects and materials make up urban environments. These varying forms of matter interact with radiation differently. For example, a snow-covered sidewalk scatters and reflects a great deal of energy, but an asphalt road is better at absorbing it. This phenomenon can be described by a material's **albedo** (ρ). The albedo of a material is its relative reflectivity, the ratio of reflected radiation to the incident amount. This nonstatic, unit-less value between 0 and 1 can change over the course of a day and with radiation wavelength. It can be further broken down

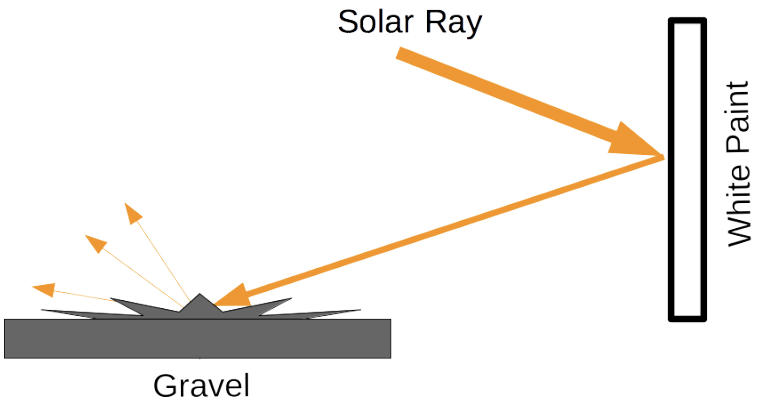


Figure 2.6: White paint has a high specular albedo (ρ) compared to many urban materials. Gravel's albedo is slightly less, but almost entirely diffuse.

into both diffuse and specular components. Snow has a higher albedo of around 0.8-0.9, while asphalt is 0.04-0.12. Note that the specific use of the term albedo varies in literature. It can be used for total Earth reflectance in mesoscale models, to describe ratio of incoming radiation to outgoing. However, since the focus of QES is on microclimate modeling, albedo is used to describe the reflectance of smaller, discretized surfaces within the domain.

Any energy that is not reflected is absorbed, described by its **absorptivity** (α). In simple models, this is often equal to $1-\rho$. While not currently represented in QES, some surfaces have the ability to let light pass through them, such as glass or porous materials. This is defined by its **transmissivity** (τ). In addition to a material's ability to absorb and reflect radiation is its ability to emit it. The **emissivity** (ϵ) describes an objects ratio of emitted thermal energy to that of a perfect emitter (i.e. black body) that has an emissivity of 1.0. Soil has a higher emissivity of around 0.94, while polished metallic surfaces emit less with an emissivity commonly lower than 0.1.

2.3.3 Physical Properties of Vegetation

Light energy is attenuated as it passes through vegetative canopies according to Beer-Lambert's Law [45]. Beer-Lambert's Law relates the attenuation of light to the properties of the material and traversal distance. That is, for an initial light intensity E_0, some absorption coefficient κ, and traversal distance b, the amount of energy left over after it passes through the vegetative canopy is $E_0(e^{-\kappa b})$.

It should be noted that following description of vegetative and radiative interactions is formed from research by Bailey, et. al. [2]. Refer to this work for more information on the origination of these physical properties.

The absorption coefficient is dependent on the probability that light will be intercepted and the vegetation's ability to absorb that light. This probability can be computed from its **leaf area density** (LAD) or **leaf area index** (LAI), the projection of light-intercepting surfaces in the direction of the light, as well as vegetation's **absorptivity** (α). LAD is used with respect to a volume of leaves and LAI describes its outer surface. Plants often grow towards areas of intense sunlight to maximize these values and its surface exposure to solar energy. These terms are not exclusive to tree-like vegetation and can be applied to many types of plants [29].

In addition to absorption, plants may scatter incident radiation in nonuniform directions. But, some plants have epicuticular wax to protect the surface from weather and moisture loss. This wax has the ability to specularly reflect radiation, in addition to diffuse scattering. We can describe these effects by assigning plants a **reflectivity** (ρ). Unlike opaque surfaces as they are represented in QES, energy that is not absorbed, scattered, or reflected is transmitted through the volume according to its **transmissivity** (τ). We can express these quantities with the formula $1 = \alpha + \rho + \tau$.

The size and shape of each plant, as well as the physical characteristics has a wide

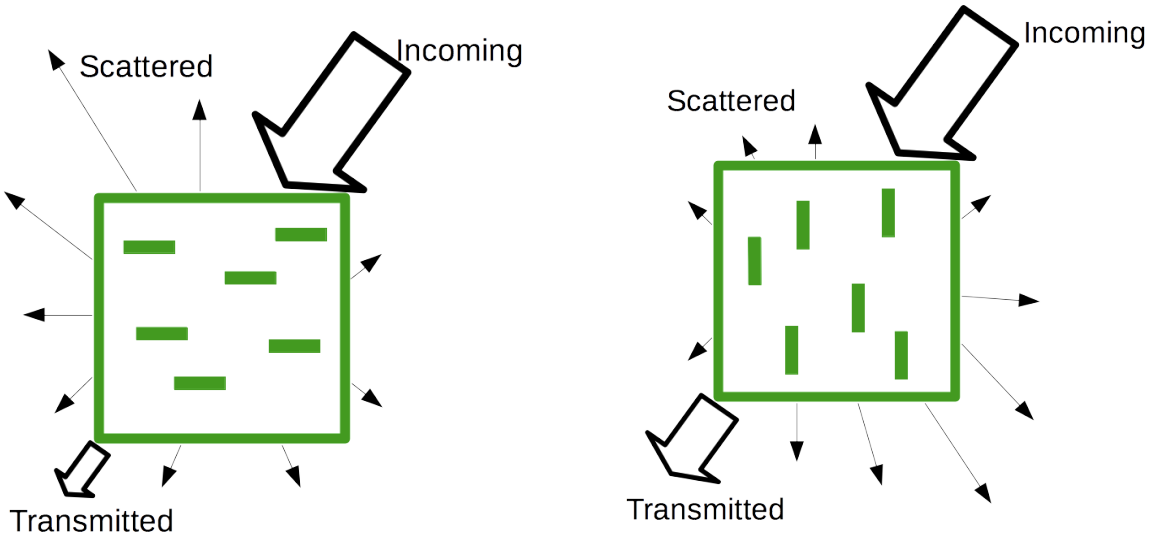


Figure 2.7: Energy is scattered in the a direction depending on its incoming angle and leaf distribution. Some energy may make it all the way through. This figure illustrates the direction of scattered and transmitted energy for horizontal (left) and vertical (right) distributions.

range of variation. These complexities pose a great challenge to modeling vegetation in urban environments. Unlike common urban materials, the properties of vegetation are dynamic and change with season, environment, and regular human intervention.

2.3.4 Simulating Radiation

The Q^* term of the energy balance equation (2.1) is the total incoming and outgoing radiation. The complete equation is shown in equation 2.2. $R^{\downarrow}_{par}$, $R^{\downarrow}_{nir}$, and $R^{\downarrow}_{\ell}$ is incident PAR, NIR, and longwave radiation. $R^{\uparrow}_{par}$ and $R^{\uparrow}_{nir}$ is reflected PAR and NIR, while $R^{\uparrow}_{\ell}$ is emitted longwave. The full net radiation equation can be defined as:

$$Q^* = R^{\downarrow}_{par} - R^{\uparrow}_{par} + R^{\downarrow}_{nir} - R^{\uparrow}_{nir} + R^{\downarrow}_{\ell} - R^{\uparrow}_{\ell} \quad [\mathrm{Wm}^{-2}]. \tag{2.2}$$

Previous iterations of QES loosely approximated surface net radiation with sun view factor (F_{sun}), sky view factor (F_{sky}), and wall view factors (F_{wall}), which are

explained in greater detail in section 3.3. Sun view factor represents the fractional amount a surface is exposed to the sun. Sky view factor is the fraction of the surface that is unobstructed from the sky. Wall view factors are the fraction of how much two surfaces see each other. Obtaining view factors was done using Monte-Carlo ray tracing, in which rays were launched from surfaces into the environment and their end points recorded. The exact calculations used for these view factors is described in section 3.3. An approximation of the amount of shortwave energy could be determined from F_{sun}. Longwave irradiation from the atmosphere and night time cooling was approximated with F_{sky}. Incoming longwave from terrestrial sources was approximated using F_{wall}. Essentially, this approach modeled radiation in the *reverse* direction.

With the introduction of vegetation, the net radiation at a surface could no longer be efficiently modeled this way. View factors can not easily capture the directionally dependent scattering and absorption of energy. The rays must originate from their respective sources and be augmented by objects as they traverse through the domain. This approach correlates to the true physical operations of radiation, allowing for more detailed and accurate simulations. The implementation details of this approach are specified in section 3.4.

2.4 Graphics Processing Units

A Graphics Processing Unit (GPU) could be considered a mini-supercomputer. GPUs consist of hundreds to thousands of cores that are designed for parallel execution. They have seen explosive growth in the last ten years due to consumer demand for sophisticated video game graphics. However, recently there has been a trend to adapt this hardware for general purpose computing (GPGPU) [32]. GPGPU is rep-

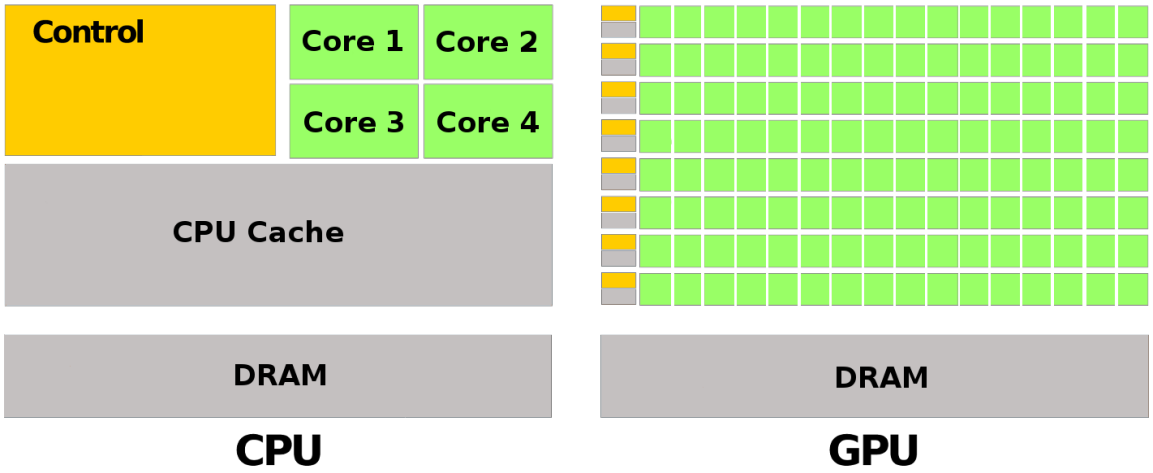


Figure 2.8: CPUs and GPUs are architecturally different. CPUs may contain a few high performance cores, while GPUs consist of larger arrays of weaker cores. "CPU-GPU" by NVIDIA is licensed under CC 3.0.

resented in numerous fields, such as in fluids simulation, computational mathematics, and visualization. Many algorithms that benefit from a highly parallel environment can utilize the GPU.

To develop GPGPU software we define **kernels**, functions that operate on the threads of the GPU. Many GPGPU libraries can be used to facilitate the development of kernels, namely NVIDIA CUDA [28] and OpenCL [44]. Kernels developed in CUDA will only operate on NVIDIA brand GPUs, while OpenCL has cross platform capability.

A kernel's execution is based on the single-instruction multiple-data (SIMD) computation archetype [22]. In SIMD, the same function executes on a series of elements in a vector of data and can do so in parallel. SIMD computing was adopted in the early 1970s by vector supercomputers such as the Illiac IV [4] and CDC STAR-100 [17]. Many modern central processing units (CPUs) include an extended set of instructions for SIMD computing, such as Streaming SIMD Extensions (SSE) [35] which can be used on both Intel and AMD processors. For SIMD algorithms, however, consumer-

class GPUs exceed the performance capabilities of CPU-based SIMD processing on single node systems [32]. Due to their low cost and high availability, GPUs are an attractive platform for SIMD algorithms.

Architecturally, NVIDIA brand GPUs consist of an array of streaming multiprocessors (SMP). Each SMP contains an array of compute cores. The specific architecture and layout of SMP and cores vary with compute capability. When a kernel is launched, its required resources are copied to an entire SMP as a **warp**, where 32 threads act on data simultaneously. For these operations to occur, input and output data must be copied between the **host** and **device**. The host is the CPU which controls the launches of kernels on the device, or GPU. This host-device copy is a computationally expensive procedure, and can sometimes be the bottleneck of GPU accelerated applications [48].

Single precision floating point format is often used when doing calculations on the GPU. This increases the computational efficiency of kernel execution. However, double-precision floating-point format can be used when necessary for values outside the capability of single precision. This requires a compute capability of 1.3 or higher, which includes nearly all NVIDIA GPUs made after 2009.

The challenges of GPU computing fall upon acceleration strategies that novice programmers may not be familiar with. Unlike CPU computing, it is expensive for a GPU thread to retrieve data from global memory. As a form of stream processing, it does not rely on sophisticated caching of memory as serialized CPU processing does. In addition, there is much less memory available to kernels, both locally and globally. Each thread must manage its own limited program stack. Debugging critical errors without the help of common CPU profiling tools can be a challenge. If threads are doing too much work they may be terminated by operating system *watch dog* timers. These issues and more require a deep investigation into the APIs for developing

Figure 2.9: Ray tracing can be used to generate sophisticated images that contain reflections, caustics, and global illumination. "Glasses, pitcher, ashtray and dice" by Gilles Tran, public domain.

applications for the GPU.

2.5 Ray Tracing with NVIDIA OptiX

Recall that sky view factors and radiation can be simulated with a technique called ray tracing. In this approach, a ray is defined by its origin and direction, and objects are defined by their geometrical shape. To determine if the ray will hit something, ray-surface intersection tests are computed for every object in the domain, called a **scene trace**. A common strategy to accelerate this algorithm is to lower the amount of ray-surface intersection tests, which can be done with bounding volume hierarchies (e.g. figure 2.10) and other techniques.

It is often used for sophisticated image synthesis, as seen in figure 2.9. To do this, an image plane is sampled by launching rays in toward the virtual domain from each

pixel. The ray may bounce off reflective surfaces or pass through caustic materials. The end result is radiance that is used to determine the pixel color. Ray tracing is an active area of formal research in computer graphics. Many different techniques use ray tracing to compute illumination of a virtual environment, such as path tracing, photon mapping, and ray marching algorithms such as the one described.

Ray tracing has the feature of being extremely parallelizable. That is, each ray's scene trace can operate independently. This makes ray tracing a prime candidate for GPU acceleration. However, complexities in recursive ray tracing require sophisticated computation to be effective on the GPU.

In 2010, NVIDIA released OptiX, the general purpose GPU ray tracing engine [33]. It consists of two APIs for both the host and the device. Developers are able to create their own launch and intersection kernels, as well as define the primary interactions of a ray tracer. OptiX will compile this information and efficiently perform the scene trace on the GPU.

The primary mode of communication between the host and the device is through the OptiX context, which operates as an instance of the engine. Input and output is stored as temporary arrays, called **buffers**. A buffer can be labeled as read only

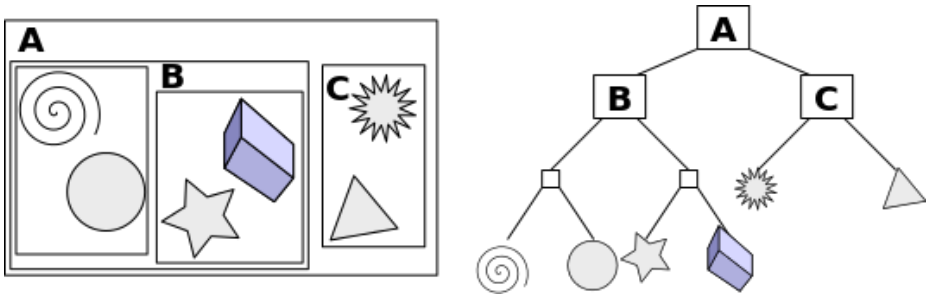


Figure 2.10: A bounding volume hierarchy wraps neighboring objects in groups (left) to form nodes of a tree (right). When the tree is traversed, whole branches can be excluded from ray intersection tests. "Example of bounding volume hierarchy" by Schreiberx is licensed under CC 3.0.

(INPUT), write only, (OUTPUT) or readable and writable (INPUT_OUTPUT) by the device.

QES relies heavily on the optimizations provided by NVIDIA OptiX. Many of the strategies, program flow, and data containers are designed to interoperate with OptiX. It is used for both computing view factors and simulating radiation transport.

2.6 Challenges of Putting it All Together

Many details of radiative exchange in urban environments are presented in section 2.3. Developing an efficient computational method of these details is a primary goal of this research due to its dominant effect on urban microclimate. Other models, such as a more sophisticated land surface model or air and wind transport models are needed for a more accurate simulation of equation 2.1. Developing these models can be difficult, because many of there functions must be tested and validated in isolation. That is, environmental properties or specific terms of the energy balance equation are set as a control. Once validated, they must be coupled with other models and verified further. Taking advantage of high performance computing techniques like GPU computing just further increases the difficulty. The necessary ablation of these scientific models represents the key challenge of developing a system for coupled microclimates. Model implementations must be isolated at some times, and highly coupled at others. Even once a model is validated, it is not always necessary to simulate it. If a test case only requires computation of sky view factors and not surface temperatures, it is a waste of resources to simulate the entire land surface model or equation 2.1.

It is clear that what is needed is a dynamic framework for handling the arbitrary coupling of microclimate models that eases technical hurdles of acceleration, memory

management, program organization, and GPU computing. An environment where models can be developed independently and easily coupled.

3 Implementation

QES currently contains six separate modules, four of which are designated a specific section of this chapter. In these sections, the models, tools, and data it encapsulates are described. Test cases are detailed in chapter 4. Other sections of this chapter outline shared functionality between modules, such as sampling and acceleration techniques. Following the modular programming design approach, each module has specific tasks that it is responsible for. However, some modules can be coupled with others to provide them with input. These modules include:

1. `QESCore`: The *core* of QES which contains necessary tools for developing models. All other modules rely on `QESCore` to handle the primary functions of microclimate modeling. These functions are discussed in section 3.2.

2. `QESViewfactor`: Computes sun, sky, and wall view factors for every surface in the domain. These computations are described in section 3.3.

3. `QESRadiant`: Computes the exchange of radiant energy between the sun, surfaces, vegetation, and the atmosphere. Discussed in section 3.4.

4. `QESLSM`: Solves a simplified energy balance equation for every surface to compute temperatures. When coupled with `QESRadiant`, the results of the radiation exchange are automatically used as input to compute surface temperatures. Otherwise, the user must specify the radiation values manually. Details are given in section 3.5.

5. `QESTransport`: Computes diffusion of heat and moisture between surfaces and nearby volumes of air due to turbulent wind flow. This module was developed by Briggs [6].

6. `QESGUI`: The graphical user interface module that can display output from any of the other QES modules. Implementation details are given in [15].

3.1 System Pipeline

There is certain functionality that is required for every module of QES such as verification of user input and construction of the virtual environment. These activities are handled by `QESCore`. It is also responsible for controlling the system pipeline, that is, the flow of execution state. How each state is invoked by the API is described in section 4.1.1. There are four main states of program execution that operate in specific order and cannot progress to a previous state:

1. Setup: CPU resources are allocated and simulation settings are defined. The user will specify the domain to operate on, as well as which models and modules to use.

2. Initialization: GPU resources are allocated based on the models specified. The scene is built and cannot be changed beyond this point. Input is verified to avoid common errors and defaults are assigned to parameters not specified. Only model-specific input, such as ray samples or run time settings, can be modified beyond this state.

3. Simulation: The models are executed. This state can return to itself in which new simulations are executed if input is modified. The program remains in this state until the instance of the program is exited.

4. Termination: CPU and GPU resources are deallocated, cleaned up, and the program terminates.

3.2 QESCore

3.2.1 Context

The `Context` is an instance of a QES system. The `Context` provides means of allocating and deallocating computational resources, initialization of models and input parameters, running the simulations, and controlling the flow of execution state. In a sense, the `Context` is the heart of QES. It is recommended to have only one running `Context` on a machine at time. By extension of NVIDIA OptiX, the `Context` is not guaranteed to be thread safe. In addition, the `Context` attempts to utilize all GPU resources allotted to it. Multiple `Contexts` for a single GPU results in GPU resource competition and is not recommended. However, if different devices are assigned to different `Contexts`, it is possible to have more than one running `Context` at a time. Note that the QES `Context` is not the same as an OptiX context. The two are separate entities.

3.2.2 Resource Management

QES has multiple tools that act as a control mechanism to allow models to communicate and users to set and query system settings. These isolated system components also attempt to handle sophisticated tasks behind the scenes and prevent the redundant storage of data. Tools of primary importance include:

BufferTracker

Buffers are temporary arrays of elements that reside on the device. They are the primary method by which the GPU stores input and output. In order to allocate this memory in OptiX, a use must declare a unique handle name, number of elements, element type, and the buffer type. Refer to the OptiX Programming Guide for more details on these requirements [27]. The `BufferTracker` is a wrapper for the underlying commands of host-buffer interactions in OptiX. It supplies functions not directly available in the OptiX API, such as copying the contents of one buffer to another or retrieving the contents of device-local memory. In addition, all buffer copies (such as setting or retrieving data) is parallelized with OpenMP. It also inspects the buffer it is acting upon, adding additional debugging statements and operations. For example, it may warn a user attempting to set the values of an `OUTPUT` buffer. These extra operations are hidden from the caller to provide easier interaction. More details on some of the unique functionality of the `BufferTracker` are in section 3.7.3.

```
// This example sets the first 100 values of buffer "input_buff"
// to 1.f. g_buffTracker is a pointer to the globally shared
// BufferTracker, owned by the context.
std::vector<float> input_values( 100, 1.f );
bool success = g_buffTracker->setBuffer<float>( "input_buff", ↩
    input_values );
```

VariableTracker

Often models need to communicate certain variables to each other, such as user-defined system settings and sampling values. Global variables are typically used to facilitate top level variables that are modifiable by any class in the application. However, true global variables pose issues with program flow and readability.

The `VariableTracker` is a host controller for global variables. Users and models can set and retrieve values of different literal types. Pseudo-global variables can be created at any scope dynamically, allowing two way communication in the global program scope.

Sometimes it is appropriate to restrict modification of certain variables. For example, consider the case where a buffer of a static size is created based on an input variable. The module that creates the buffer can then *lock* the variable through the `VariableTracker` so that no further changes can be made. The `VariableTracker` also centralizes program variables, so that their values may be dumped to a text file for debugging or profiling purposes.

```
// This example checks if a global variable has been set.
// g_varTracker is a pointer to the globally shared
// VariableTracker, owned by the context.
bool var_set = g_varTracker->exists( "some_variable" );
```

ProgramTracker

OptiX programs are kernel functions executed on the device invoked by a **launch**. Such programs are *ray launch* (entry points), *any/closest hit*, and *miss*, to name a few. Before entry points can be declared in OptiX API, the total number of ray launch programs must be counted up and given a unique ID. Since many models *don't know about other models*, the `ProgramTracker` can satisfy these requirements without model knowledge. It will accumulate the details of OptiX programs during SETUP and create them in INITIALIZATION. The models can also add new any/closest hit functions, new ray types, and new object intersection kernels to the system via the `ProgramTracker`.

In addition to managing and creating programs, the `ProgramTracker` attempts

to handle common errors that may occur during run time execution. For example, a kernel may create too much local memory and overflow the thread stack. This will throw an OptiX *stack overflow* error, and the kernel will have failed to execute. The `ProgramTracker` will recognize this error, increase the per-thread stack, and re-execute the kernel. If an error can't be handled, it will be printed to the screen.

```
// This example launches the OptiX program "launch_rays"
// using rtContextLaunch2D from the OptiX API. It also checks ↩
    for
// common exceptions and will handle certain errors.
// g_progTracker is a pointer to the globally shared
// ProgramTracker, owned by the context.
bool success = g_shared.progTracker->launch( "launch_rays", dx, ↩
    dy );
```

SceneTracker

Handling all of the elements that make up an urban environment is a large task. There are building and vegetation geometries, material types, geographic location, and more. The `SceneTracker` is a host tool for managing these attributes. Through the `SceneTracker`, a user can specify the environment which maintains ownership of scene geometry data. Refer to chapter 3.2.3 for more information on how the domain is constructed through the `SceneTracker`.

```
// This example loads a QUIC project and supplementary XML file.
// g_sceneTracker is a pointer to the globally shared
// SceneTracker, owned by the context.
std::string myProj = "/QUICProjects/myProject/myProject.proj";
std::string myXML = "/QUICProjects/myProject/myProject.xml";
bool success = g_sceneTracker->initScene( myProj, myXML );
```

InputTracker

Some models must be driven with empirical data. For example, without a model to compute the air temperatures, the land surface model must refer to measured data as input. With many models requiring different input at different stages of execution, relying on the user to manage these inputs can be overwhelming. The `InputTracker` provides a centralized location in which a user can load *standardized data*. Models will request this data as needed during execution. For more information on this data and how this process works, refer to chapter 3.2.5.

```
// This example loads a Surface Weather Map XML file which will
// be used as input by the models.
// g_inputTracker is a pointer to the globally shared
// InputTracker, owned by the context.
std::string mySWMXML = "/SurfaceWeatherMaps/Utah2014.xml";
bool success = g_inputTracker->loadSWMXML( mySWMXML );
```

SunTracker

The energy of the sun drives the urban microclimate. The sun's position relative to a geographic location (i.e. solar vector) varies with time. The `SunTracker` is a tool that can compute the direction of the sun from a given latitude, longitude, date, and time. The `SunTracker` also acts as controller by which a user can change the simulation time or geographic location. The model used for computing the solar position is by Blanco-Muriel et. al. which reports a higher accuracy compared to many other models [3].

The `SunTracker` also contains various methods for handling a wide range of input. Time can be set as Local or UTC, latitude and longitude can be set in decimal form or UTM coordinates. The date can be set as Julian Day, Day Number, or Gregorian.

```
// This example sets the time and date. When a simulation is run,
// the solar position will be calculated with the updated information.
// g_sunTracker is a pointer to the globally shared
// SunTracker, owned by the context.
int year = 2014; int month = 1; int day = 1; // January 1st, 2014
int hour = 12; int minute = 0; int second = 0; // Noon
g_sunTracker->setDate( year, month, day ); // Gregorian date
g_sunTracker->setTimeLocal( hour, minute, second ); // Local time
```

3.2.3 Domain

Before running any simulations, the user must specify the buildings, surfaces, trees, sensors, and other objects that exist in the simulated world. Aircells, or discretized cubes of air, may also be constructed if needed to model turbulent transport. This collection as a whole is often referred to as the computational **domain** or **scene**. In order to input this information, a user must interact with the `SceneTracker` tool during the SETUP phase of program execution. After INITIALIZATION, the geometrical representation of the scene becomes constant and cannot be changed.

The scene can be specified in three ways:

1. **XML Document**

   An XML Document that can be loaded by the `SceneTracker` must specify several parameters, as well as provide a list of buildings, vegetation, their physical properties, and sensors. A minimal example can be seen in appendix B.3.

   While an XML file can be used to specify an entire domain, it can also be used to supplement a different source. In this case, all buildings and trees from the XML file are loaded into the scene in addition to buildings and trees from the

other source. An example of this is shown in figure 3.1. The `SceneTracker` makes no attempt at avoiding overlapping or intersecting buildings and trees.

2. **Hard-Coded Test Case**

   A typical way to test the system or define simple domains is by loading hard coded test cases, or programmatically defining it. In the former, a user can specify a test case that will be generated by code. These test cases were implemented to test specific functionality of models and publication validation. A list and description of these test cases can be seen in appendix C.1.

   To have a user create a scene themselves, they must only populate the lists of data structures that define the scene. These are `std::map` tables owned by the `SceneTracker`. A user can simply add additional buildings, vegetation, and sensors to these tables during the SETUP phase. Factory classes are provided to

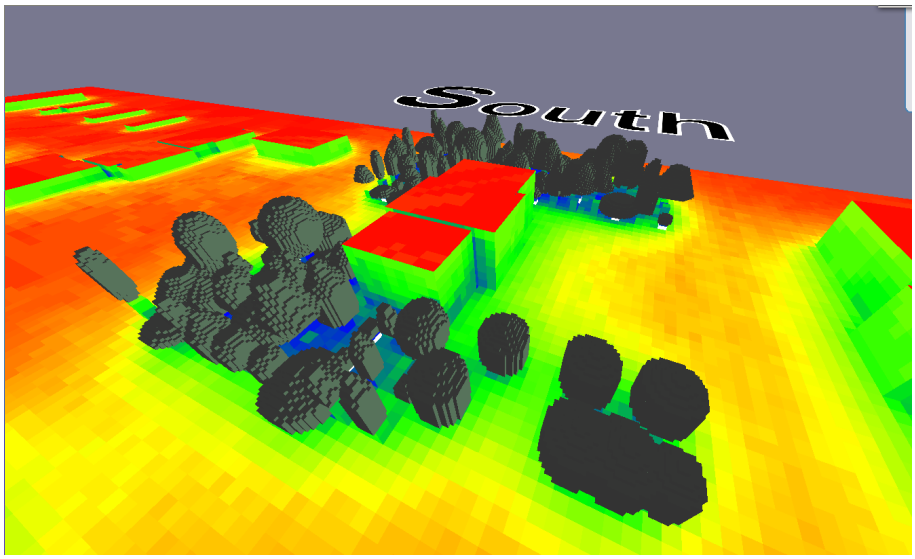


Figure 3.1: QES visualization of sky view factors for Washington Park in downtown Salt Lake City, Utah. The QUIC Project that was loaded did not include trees, so an XML file was used to supplement the domain construction. Parks and other vegetative areas can be simulated more accurately.

simplify this process. An example can bee seen in appendix B.2.

3. **QUIC Project**

   QUIC Projects are urban domains created with the QUIC City Builder application [7]. These projects may contain output generated by the various QUIC applications, such as wind vector and velocity fields. A QUIC Project will define all buildings and bulk vegetation (canopy) in the domain, as well as UTM coordinates and world dimensions. Like hard-coded test cases, QUIC Projects can be supplemented with XML files to describe additional buildings, vegetation, and sensors.

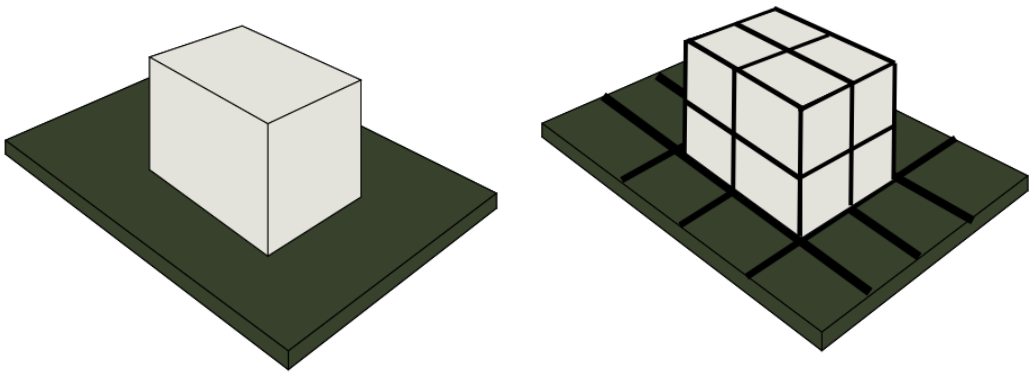
The end result of any kind of scene construction is a list of buildings, vegetation, and sensors, that are fully described with specific parameters defined. In addition, special values that describe dimensions are required. These values include the dimension (in meters) of a patch, a vegetation volume, and the ground plane.

During the INITIALIZATION phase of program execution, geometry is constructed for use with OptiX. The physical representation of these objects will differ between the host and device due to usage requirements and available memory.

Buildings

Buildings are represented as grid-aligned three-dimensional boxes with positive, arbitrary dimensions. This composition stems from domains generated by the QUIC City Builder. Often a virtual building will be made up of several boxes that are placed adjacent to, or on top, of each other.

The faces of each box is further segmented into smaller two-dimensional planes called **patches**. The size of every patch is uniform and dependent on the *patch dimensions* variable declared during the SETUP phase of program execution. The



(a) A white painted building placed on top of a grassy surface.

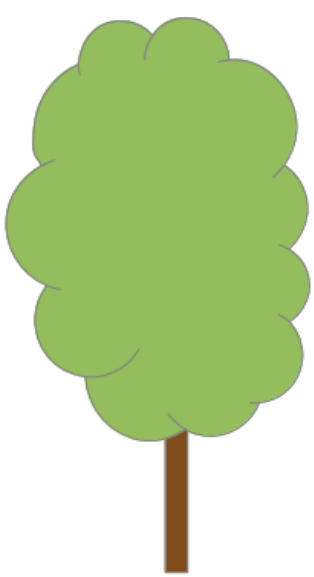
(b) A building and ground surface represented by patches.

Figure 3.2: A building and ground surface before (a) and after (b) discretization.

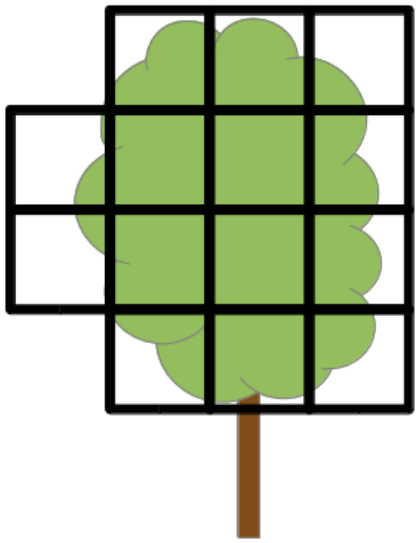
accuracy of this representation may increase with a smaller patch dimension. That is, the more patches we have per building surface, the higher the resolution and more reasonable the accumulation of energies is at a specific point. However, increasing the number of patches in the domain also increases the computational complexity and memory requirements of the system.

One special piece of geometry is the ground. Unlike buildings, the ground is specified by the *world dimensions* variable. The ground is a horizontal, two-dimensional plane that always spans the entire domain. This ground surface is used to determine the emitting plane during radiation simulation in section 3.4. Thus, if it is inappropriately specified during SETUP, necessary adjustments will be automatically applied during INITIALIZATION.

Each patch carries the physical properties of a surface described in section 2.3.2. The user has the option of defining these parameters manually, or by assigning it a predefined material type. The current list of available materials and their associated physical properties is shown in appendix A.1. Unless otherwise specified, the ground is set to soil properties, building walls are red brick, and roof tops are tar paper. The



(a) A generic tree before discretization.



(b) A representation of a tree as several vegetative volumes.

Figure 3.3: An ellipsoidal tree before (a) and after (b) discretization.

interior of buildings are not currently considered, but will be the focus of future work.

A domain can consist of one to several million patches.

Trees and Vegetation

A tree is composed of a cluster of isothermic cells that represent the leaves and branches, called **vegetative volumes**. Trees can be formed of any number of volumes of arbitrary size, so long as the physical properties of the vegetation they represent remain mostly constant within the volume. These vegetative volumes can also be used to form shrubs, weeds, dense canopies, and virtually any kind of participating media that can be expressed in volumetric form. The trunks of trees are represented as buildings with wood material properties.

Each vegetative volume has its own physical properties that designate the way it interacts with the environment, described in section 2.3.3. Each vegetation volume has a defined value for its transmissivity, reflectivity, absorptivity, leaf area density,

emissivity and leaf distribution function.

The simplest way to define a tree is to use the factory class `TreeBuilder`. In this approach, a tree is defined by various terms such as its height and crown radius. These values are inputted into the builder, which fills the space with grid-aligned vegetative volumes. There are three generic tree shapes that can be defined this way: ellipsoid, rocket, and cone. An example of this is shown in appendix B.2.

Unlike buildings and patches, vegetative volumes are not defined by material or species. Currently each parameter of a volume must be individually assigned. If any of these values is not defined, it is given a default value that does not correlate with any particular species and only acts as a placeholder.

Similar to patches, a domain can consist of zero to several million vegetation volumes.

Sensors

Virtual sensors are two dimensional planes that gather information about the simulated environment. Their purpose is to simulate real-world sensors and diagnostic tools, such as a camera photographing the sky view factor, or a pyranometer measuring solar irradiance. They can be arbitrarily positioned in the world and be of

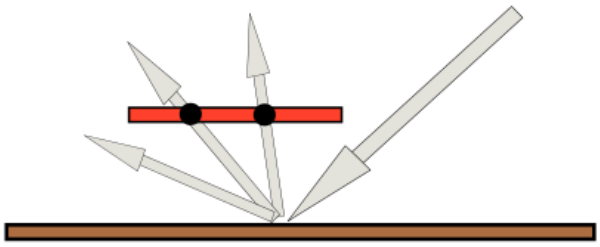


Figure 3.4: Radiation rays are scatted off the ground into the downward facing sensor. Intersections marked with • are recorded.

any size. The information obtained by a sensor depends on the models used. In `QESViewfactor`, a sensor will compute the sky view factor in the direction of its normal. In `QESRadiant`, a sensor will record (but not augment) LW, PAR, and NIR radiation that passes through its face in the direction of its normal. If a module does not define sensor behavior, it is simply ignored.

Sensors are the primary tool of validation used by QES. For example, to validate the reflectance models of the ground, a downward facing sensor is assigned a position slightly above the surface. Any energy that comes in from above the sensor is ignored, while energy that is reflected or scattered by the ground will be recorded. Figure 3.4 illustrates this example.

Air Cells

In some models the volume of air adjacent to an object must be known for effects such as the convective transfer of heat. When a wind field data set is loaded, heat is carried in the direction of the turbulent flow. This model is called a **turbulent transport model** (TTM). The TTM is currently being developed as a part of the `QESTransport` module for QES by Briggs [6]. To facilitate this model, the area outside buildings and vegetation canopies is discretized into volumes, called **air cells**.

Each air cell has numerous physical properties associated with it. These include the wind velocity, wind direction, temperature, moisture content, and others. Wind fields are loaded from QUIC data to populate the domain with turbulence data.

3.2.4 Models

Recall that a model is a physical and numerical representation of observed phenomena. This section will explain how these models are represented syntactically

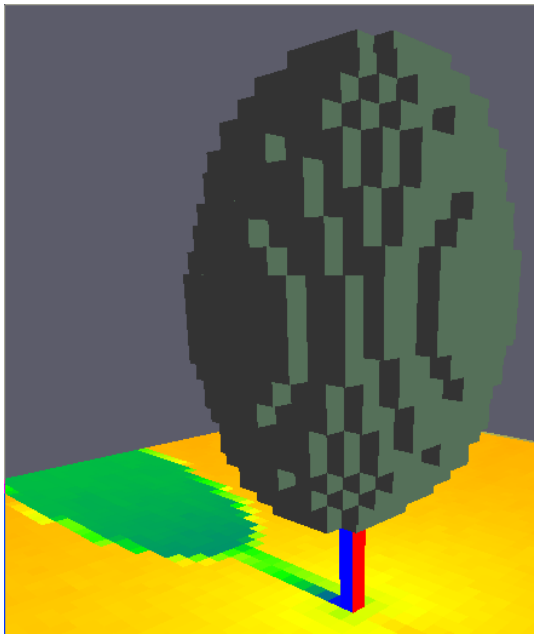


Figure 3.5: An ellipsoid tree being rendered with `QESGUI` after it has been discretized into vegetative volumes. Solar energy is attenuated as it passes through the crown, creating a shadow on the ground.

within QES. A model is usually represented as a function or class that handles the simulation and computation, encapsulated by a QES module. It requests data to be allocated via resource management tools, performs computations, and stores output. In order for a model to be defined within QES, it must allocate its resources with these tools. However, to have access to shared resources it must be **joined** with the system. In this step, the model or module is granted use of the tools described in section 3.2.2. From there, the `Context` will safely regulate access to its resources and tools. A model or module is joined to the context with the following function call:

```
qes::SomeModel model;
context.joinModel( &model );
```

To join a model to the context, two specific functions must be defined by the model class. First, an initialization function that is executed during the INITIALIZATION phase in which buffers and variables can be allocated. Prior to this function, a model does not have access to the shared tools of QES. A run-simulation function must also be defined which is executed during the SIMULATION phase. A model may specify its own cleanup function, but any data that is created with the shared tools, such as device memory, will be deallocated by the tool itself. An example model implementation is shown in 4.1.3.

Submodel

While not explicitly defined in the QES module, a model will often rely on other models to supply it with input. For example, the land surface model requires the radiation model to compute incoming radiation as one of its inputs. In this case, the radiation model is a **submodel** of the land surface model. Submodels do not have to be explicitly joined with the `Context`, so long as the parent appropriately initializes the resources it needs.

3.2.5 Simulation Input

It is typical for certain models to use measured values as input. For example, when computing surface temperature in the `QESLSM` we must know how much heat is lost to air. This value is based of the temperature difference, so the current air temperature adjacent to the surface must be known. If there is no model to compute air temperature it must be supplied as simulation input. This will vary with date, time, and geographic location. The object that handles loading data sets and querying these values is the `InputTracker`.

A surface weather map (SWM) XML file generated by MesoWest can be loaded by the `InputTracker`. A SWM contains a list of **observations** at a given date, time, and geographic location [18]. There can be any number of observations per file. An example SWM file can be found in appendix B.4. Each observation specifies certain atmospheric conditions, such as dewpoint temperature and atmospheric pressure.

The values of each observation are stored and indexed by its latitude, longitude, date, and time. Alternatively, a user can implement their own parser and manually add observations to the `InputTracker`. The only requirement is that each variable name is the same as a SWM observation.

During the simulation phase, several models will request an observation and use its contents to drive their operations. Because the date, time, and location of the current simulation may differ from the available observations, the `InputTracker` will attempt to find the *nearest* observation that has been loaded. Nearest is determined by computing distances and selecting the minimum of its geographic location, year, day number, and time, listed in order of priority.

Future work may be to automatically retrieve SWM data from online sources for a given test case.

3.3 QESViewfactor

Recall in section 2.3.4 that view factors were explicitly used in previous iterations of QES to approximate net radiation. While this is no longer the case, view factors still represent a useful measurement and analysis tool. Because of this, much of the work done by Halverson [15] and Clark [11] was rewritten for QES. The following section describes the implementation details of sun view factor F_{sun}, sky view factor F_{sky}, and wall view factors F_{wall}. View factors are expressed as numeric values that

range from zero to one.

Sun View Factor

Sun view factor refers to the ratio of a patch that receives direct solar radiation. To compute this, N_s collimated rays are launched from a patch in the direction the sun. The origin of these rays are evenly distributed across the area of the patch, then jittered to reduce aliasing effects, described in section 3.6. If a ray does not intersect anything, a value of $1/N_s$ is added to that patch's F_{sun}. We can define this summation as:

$$F_{sun} \approx \sum_{i=1}^{N_s} \frac{\varphi_i}{N_s}. \tag{3.1}$$

Where φ_i is the boolean indicator of destination. That is, if the i^{th} ray reaches the edge of the domain without intersecting anything, $\varphi_i = 1$, otherwise $\varphi_i = 0$.

Sky View Factor

The sky view factor may be defined as the ratio of a patch that receives radiation from the sky to the radiating hemisphere. To compute this, N_ℓ rays are launched outward in a hemisphere about the center of every patch. Sampling the hemisphere as a function of the number of rays per patch is discussed in section 3.6. The sky view factor is then computed by summing up the number of rays that did not intersect an object and dividing by the total number of rays launched per patch. Using the same boolean indicator as sun view factor and ray zenith angle θ_z, this can be formulated:

$$F_{sky} \approx \sum_{i=1}^{N_\ell} \frac{2cos(\theta_z)\varphi_i}{N_\ell}. \tag{3.2}$$

OptiX Any-Hit

For sun and sky viewfactors, only a boolean indication of ray-object intersection is required. The identity of the object intersected is inconsequential. Thus, when defining the intersection kernel we use an OptiX Any-Hit program. For Any-Hit programs, the OptiX engine does not attempt to identify the closest object to the origin of emittance, providing a faster scene trace. Once an intersection is determined, the ray's life will end and no further intersection tests are computed for that ray.

Wall View Factor

The wall view factor can be defined as the fraction a patch that receives radiation from another patch to the entire radiating hemisphere. Hence, each patch has a number of wall view factors equal to the number of patches in the domain. Similar to sky view factor, N_w rays are launched from the center of a patch in an outward hemisphere. If the i^{th} ray hits the j^{th} wall, a value of $2cos(\theta_z)/N_w$ is added to the wall view factor, with ray zenith angle θ_z.

Instead of storing each value $F_{wall,j}$, wall view factors are computed one patch at a time. Device memory where these values are stored is reused in subsequent ray launches, while accumulated output for every patch copied and stored on the host. In addition, only non zero wall view factors are stored.

For a boolean indicator φ, with the i^{th} ray hitting patch j, $\varphi_i = 1$, otherwise $\varphi_i = 0$:

$$F_{wall,j} \approx \sum_{i=1}^{N_w} \frac{2cos(\theta_z)\varphi_i}{N_w}. \tag{3.3}$$

Unlike sun and sky viewfactors, the knowledge of which object is closest along a ray trajectory must be known. Therefore, it requires OptiX Closest-Hit programs.

3.4 QESRadiant

To simulate the transfer of radiation from one medium to another, rays are emitted from patches, vegetative volumes, the sun, and the atmosphere, with an initial energy. When the ray intersects an object such as a patch or vegetation volume, a fraction of that energy is absorbed, reflected, and scattered. The amount of energy that is absorbed, reflected, or scattered depends on the wavelength of the radiation being simulated, as well as the physical properties of the object. It is important to note that while QES includes formulas for many components of radiation transport, all or some of these calculations can be easily replaced with user-provided functions or forced data.

Radiative flux is denoted R and N is used for variable numeric sampling, such as the number of rays per patch. The down directional ($\downarrow$) is used to indicate incident energy, while ($\uparrow$) is used for radiation emitted by the object. E is used to denote the energy of an individual ray in Watts.

3.4.1 Net Radiation

To reduce the memory overhead of `QESRadiant`, only the total net radiation of solar energy is stored during the simulation. That is, the difference of incident radiation to reflected is recorded in memory as absorbed (net) solar energy. Incoming and emitted longwave energy are still recorded as separate terms due to their individual use by `QESLSM`. Therefore we can redefine the net radiation equation 2.2 as:

$$Q^* = R_{par,net} + R_{nir,net} + R_\ell^\downarrow - R_\ell^\uparrow \quad [\mathrm{Wm}^{-2}] \tag{3.4}$$

Longwave Notation in Equation 3.4

The notation of equation 3.4 may be confusing because the computation of `QESRadiant` uses an atypical formulation of Q^*. Here we define the difference of incident ($R^{\downarrow}$) solar radiation to reflected ($R^{\uparrow}$) as net energy, as it is often notated. However, equation 3.4 still represents longwave energy as incident to emitted intensities. This may lead the reader to assume that longwave energy is not reflected like solar energy, which is true using default longwave albedos (see table A.1). Yet, the user has the ability to change longwave albedos, resulting in additional longwave energy being reflected by a material. We do not define absorbed longwave as net energy like solar radiation due to the existence of emitted longwave energy. Including an additional term to represent reflected longwave energy may be notationally confusing. Thus, we let $R_{\ell}^{\downarrow}$ denote *absorbed* longwave energy as a special case.

3.4.2 Shortwave Transfer

Solar shortwave energy is received by patches and vegetation in two ways. Radiation that reaches the object directly from the sun is considered **direct** shortwave energy. Energy that has been scattered by the atmosphere and eventually reaches the surface or volume is **diffuse** shortwave energy. Diffuse shortwave is sometimes called scattered energy in literature, but we reserve the term **scattered** for radiation that is dispersed and reflected by surfaces and vegetation. The initialization of direct and diffuse shortwave ray energies is defined by its unobstructed flux, further discussed in section 3.4.5. The direct solar fluxes $S_{par,dr}$, $S_{nir,dr}$, and diffuse solar fluxes $S_{par,df}$, $S_{nir,df}$, can be assigned values directly, or simulated with a solar flux model, such as Monteith and Unsworth [26].

Direct solar radiation is simulated by launching collimated rays from the sun

toward the ground surface. N_s ray origins are evenly distributed horizontally per a square meter for every ground patch in the domain. That is, the emitting plane is a translation of the ground in the direction of the sun, with rays being emitted in the direction of the scene. This approach will cause inaccuracies at low sun altitudes. Patches of building walls near the ground will receive an extremely high amount of direct solar radiation, while patches near the top of the building may receive none at all. A more accurate and versatile approach is currently being developed. The amount of solar energy per area depends on the angle of emittance and is accounted for by multiplying the initial energy by the cosine of the solar zenith angle, θ_s. For patch area A, the amount of energy contained by a direct solar ray is:

$$\begin{aligned} E_{par,dr} &= S_{par,dr} cos(\theta_s) A/N_s \quad \text{[W]}, \\ E_{nir,dr} &= S_{nir,dr} cos(\theta_s) A/N_s \quad \text{[W]}. \end{aligned} \tag{3.5}$$

Diffuse solar radiation is simulated by launching N_s rays toward the domain in a hemisphere about every ground patch. Because the amount of diffuse solar radiation is stronger in the direction of the sun, a radiance distribution $N(\theta_r, \Gamma_r)$ by Harrison and Coombes [16] was applied for ray azimuth angle Γ_r and zenith angle θ_r to appropriately modify the intensity of the ray . Each diffuse solar ray will carry:

$$\begin{aligned} E_{par,df} &= N(\theta_r, \Gamma_r) S_{par,df} 2cos(\theta_r) A/N_s \quad \text{[W]}, \\ E_{nir,df} &= N(\theta_r, \Gamma_r) S_{nir,df} 2cos(\theta_r) A/N_s \quad \text{[W]}. \end{aligned} \tag{3.6}$$

3.4.3 Longwave Transfer

Nearly all terrestrial objects emit longwave radiation. To simulate this emission, rays are launched from patches in a hemisphere about its surface, and a sphere about vegetation volumes. These functions are derived from the radiation transfer model

by [2].

The total amount of energy emitted from a patch depends on its is the emissivity ϵ, Stefan-Boltzmann constant σ_s, temperature T, and area A in meters. When a LW ray of N_ℓ rays is emitted from a patch it holds:

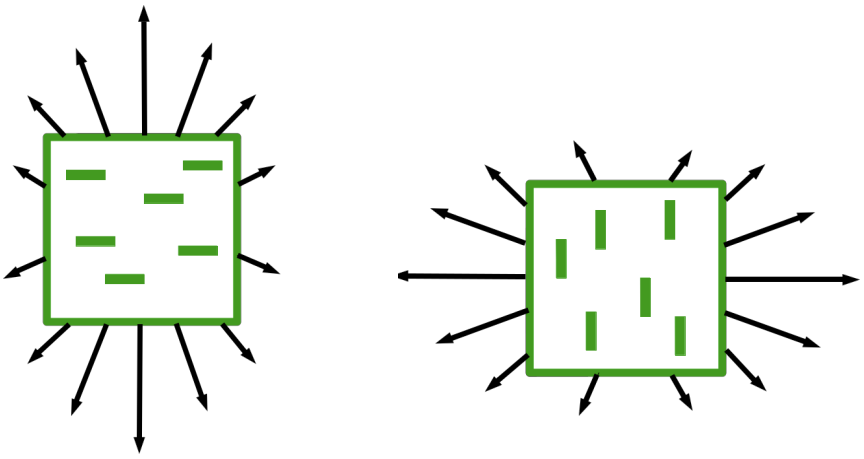
$$E_{\ell,patch} = \epsilon\sigma_s T^4 A 2cos(\theta_r)/N_\ell \quad [\text{W}]. \tag{3.7}$$

Like patches, energy emitted from a vegetation volume is dependent on its emissivity, the Stefan-Boltzmann constant, temperature, and volume. But due to the potential to self absorb emitted longwave radiation, κ is represented as the attenuation coefficient. It is the product of the leaf area density and the result of the Ross-Nilson G-function [37]. The G-function ($G(\theta_r)$) is a directionally dependent probability that the ray will be intercepted by the leaves for a given ray angle θ_r and leaf distribution. The amount of energy emitted by a vegetation volume per-ray for N_ℓ rays is:

$$\begin{aligned} \kappa &= LAD \times G(\theta_r), \\ E_{\ell,vege} &= 4\kappa\epsilon\sigma_s T^4 V/N_\ell \quad [\text{W}]. \end{aligned} \tag{3.8}$$

Note that $G(\theta_r)$ of equation 3.8 takes the distribution of leaf normals and emitted ray direction into consideration. Typically, broad leaf plants will emit more longwave energy in the direction of leaf normals despite the higher probability of self absorption. For example, horizontally-oriented leaves will emit a greater amount of energy in the vertical directions, and vertically-oriented leaves in the horizontal directions. This is illustrated in figure 3.6.

The total amount of energy emitted by patches and vegetation can be computed from equations 3.7 and 3.8 without dividing by the number of samples (N_ℓ) or applying



(a) Horizontally-oriented leaves emit more energy in the vertical directions.

(b) Vertically-oriented leaves emit more energy in the horizontal directions.

Figure 3.6: The amount of longwave energy emitted by vegetation depends on its leaf angle distribution. The adjustment of emitted ray intensity is computed using the Ross-Nilson G-function [37].

cosine weighting ($2cos(\theta_r)$). These terms can be used when computing the energy balance equation. The longwave terms used in Q^* are as followed:

$$\begin{aligned} R^{\uparrow}_{\ell,patch} &= \epsilon\sigma_s T^4 \quad [\text{W}], \\ R^{\uparrow}_{\ell,vege} &= \kappa\epsilon\sigma_s T^4 \quad [\text{W}]. \end{aligned} \tag{3.9}$$

Longwave energy also has the potential to be scattered by the atmosphere. This energy is simulated exactly like diffuse shortwave in which rays are launched in toward the domain from a hemisphere about ground patches. However, the amount of energy per square meter emitted by the atmosphere depends on the diffuse longwave flux $L_{\ell,df}$. $L_{\ell,df}$ can be specified directly, or simulated with a diffuse longwave flux model, discussed in section 3.4.5. The amount of longwave energy emitted by the atmosphere

per-ray for ground patch area A, ray zenith angle θ_r, and N_ℓ rays is

$$E_{\ell,df} = L_{\ell,df} 2cos(\theta_r) A / N_\ell \quad [\text{W}]. \tag{3.10}$$

3.4.4 Absorption, Reflection, and Scattering

Patch

Incoming shortwave and longwave is absorbed, reflected, and scattered by patches. For a ray with an initial energy E_0 and radiation type λ, and a patch with albedo ρ, the amount of energy absorbed from one ray is $E_0(1.0 - \rho_\lambda)$. Thus we can sum the absorbed energy for all rays that intersected the patch to compute $R_{par,net}$, $R_{nir,net}$, and $R_\ell^\downarrow$. That is,

$$R_\lambda \approx \frac{1}{A} \sum_{i=1}^{N_i} E_i(1.0 - \rho_\lambda) \quad [\text{Wm}^{-2}]. \tag{3.11}$$

where N_i is the number of rays that intersected the patch with an incoming ray energy of E_i.

The amount of energy that is scattered and reflected depends on the material of the patch. This property is determined by a material's specular fraction (δ_s) and diffuse fraction (δ_d) where $\delta_s + \delta_d = 1$. For example, highly reflective surfaces like glass or glossy paint would have a high δ_s and low δ_d. Conversely, sand or gravel would have a much higher δ_d and lower δ_s. The amount of energy specularly reflected is $E_0(\rho\delta_s)$ and the amount scattered is $E_0(\rho\delta_d)$. Thus, each patch is able to diffusely scatter and specularly reflect radiation. This results in a deterministic, pseudo-anisotropic scattering of radiant energy.

For specular reflections, a new ray is launched from the point of intersection. For an incidence angle θ_i, the angle of reflection $\theta_r = \theta_i$ about the surface normal is used. Scattered energy is isotropic, so new rays are launched in a hemisphere in the same

manner as emitted longwave.

Vegetation

Vegetation will absorb, scatter, and transmit radiation. In this paper, energy scattered by vegetation is isotropic. An implementation of fully anisotropic scattering has been implemented in QES, but is still being developed to meet performance requirements [31].

The amount of energy absorbed by a vegetation volume is dependent on its absorptivity α, attenuation coefficient $\kappa = LAD \times G(\theta_r)$, and the distance the ray traveled through the volume in meters b. For a ray with an initial energy E_0 and

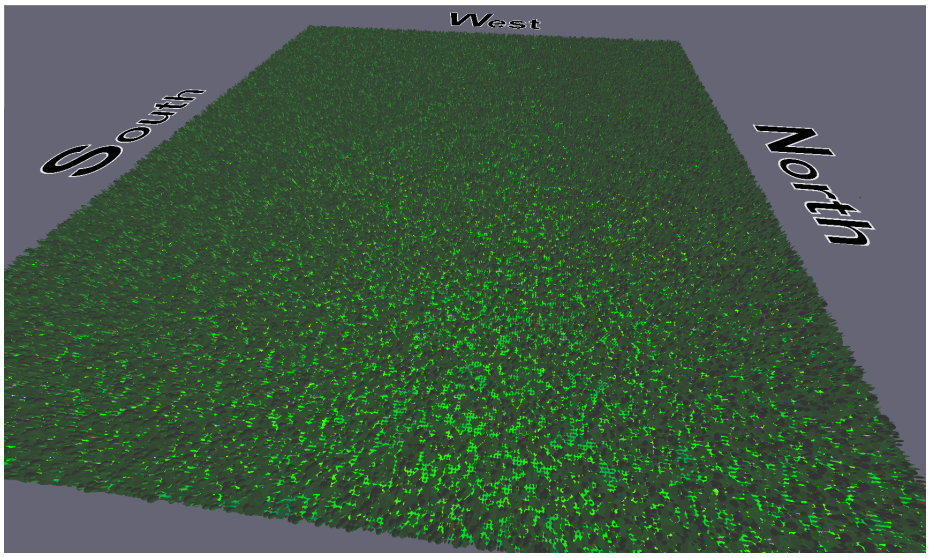


Figure 3.7: By utilizing graphics hardware and techniques for computational efficiency, `QESRadiant` is able to simulate radiation transfer in dense forests. This figure shows 115,591 randomly generated trees, with 1,020,100 patches and 1,653,065 vegetative volumes.

radiation wavelength λ, the amount of energy absorbed is

$$R_\lambda = \frac{1}{V}\sum_{i=1}^{N_i} E_i(1.0 - \mathrm{e}^{-\kappa b})\alpha_\lambda \quad [\mathrm{Wm}^{-3}]. \tag{3.12}$$

The amount of energy scattered by vegetation is dependent on the properties above, as well as its reflectivity ρ and transmissivity τ. In future models, the direction of scattered energy will also depend on reflectivity and transmissivity. However, since this model scatters energy isotropically, their sum makes up the total fraction of intercepted energy that is scattered. Thus, the amount of energy scattered is:

$$R^{\uparrow}_{scatter,\lambda} = \sum_{i=1}^{N_i} E_i(1.0 - \mathrm{e}^{-\kappa b})(\rho_\lambda + \tau_\lambda) \quad [\mathrm{W}]. \tag{3.13}$$

The rest of the energy is transmitted through the vegetation volume with its direction unchanged:

$$R^{\uparrow}_{transmit,\lambda} = \sum_{i=1}^{N_i} E_i(\mathrm{e}^{-\kappa b}) \quad [\mathrm{W}]. \tag{3.14}$$

A ray will continue to be transmitted, reflected, and scattered by patches and vegetation until it no longer carries a significant amount of energy due to scattering and absorption. Once a ray's energy drops below a predefined threshold ($10^{-6}[W]$), the rest of the energy is absorbed by the last object intersected.

3.4.5 Unobstructed Flux Models

In order to determine the amount of energy the sun emits and atmosphere scatters, we compute what is called an **unobstructed flux**. It is unobstructed in the sense its origin resides above the urban canopy and does not take the turbidity of the

atmosphere into account. This energy is then split into rays per unit meter.

Two solar flux models are currently available to use in QES. One is Monteith and Unsworth's cloudless sky radiation model for a flat surface [26], which is used to calculate total solar radiation. To distribute the energy to diffuse and direct, the model is modified based on measurements by Liu and Jordan which distinguishes the relationship between diffuse, direct, and total solar energy [23]. Another model that can only be used for daytime hours is by Spitters [43]. This simply distributes 23% of the solar constant to diffuse, 77% to direct, and should only be used during daytime hours. In either flux model, 50% of the energy is considered PAR, and the other 50% is NIR. This ratio is taken from table 5.1 in [26].

One longwave flux model is implemented in QES from Yang and Li, in which $L_{\ell,df}$ is computed from the dewpoint temperature, dry bulb temperature, and sky emissivity [47]. The dry bulb temperature and dewpoint temperature are read in as inputs from the `InputTracker` and used to calculate sky emissivity using formulas derived by Chen, Clark, Maloney, Mei, and Kasher [10].

3.5 QESLSM

`QESLSM` is a module that computes the energy balance equation (2.1) for every patch in the domain. The current implementation can be considered a *simplified* LSM in that many of the terms are computed as rough approximations. This class may also be considered a place holder to provide current researchers a start on developing a more sophisticated and realistic LSM. The methods by which the energy balance equation can be used to solve for temperatures will be discussed in this section. Recall equation 2.1:

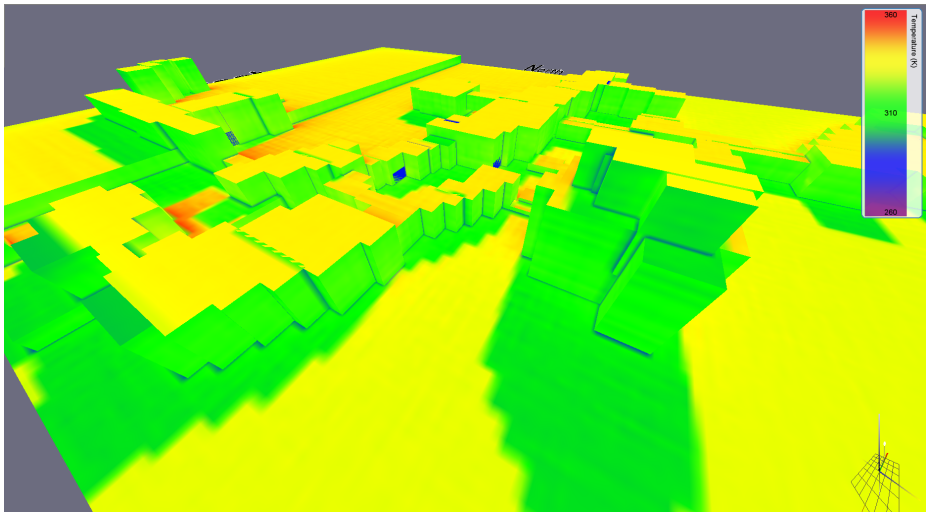


Figure 3.8: The LSM can be used to compute surface temperatures for the University of Minnesota Duluth.

$$0 = Q^* - Q_h - Q_g - Q_e.$$

In order to solve this equation, a value is computed for each of the terms in this equation. However, for the purpose of simplification, latent heat flux Q_e is assumed zero. If `QESRadiant` is also joined with the `Context`, then its output (Q^*) will be used as simulation input. Otherwise, this must be supplied by the user.

3.5.1 Expansion of Terms

The net radiation of a surface (Q^*) can be computed as an expansion of equation 3.4

$$Q^* = R_{par,net} + R_{nir,net} + R_\ell^\downarrow - (\epsilon\sigma_s T_s^4) \quad [\mathrm{Wm}^{-2}]. \tag{3.15}$$

Where $R_\ell^\downarrow$ represents longwave radiation that was absorbed by the surface, ϵ is surface emissivity, σ_s is the Stefan-Boltzmann constant, and T_s is the temperature of the surface in kelvin.

For surface temperature T_s and air temperature T_a, sensible heat flux (Q_h) is defined as

$$\begin{aligned} \alpha_c &= 11.8 + 4.2(Vc), \\ Q_h &= \alpha_c(T_s - T_a) \quad [\mathrm{Wm}^{-2}]. \end{aligned} \tag{3.16}$$

The convective heat transfer coefficient α_c is defined according to [38] with wind velocity Vc. T_a and Vc can be obtained from the Surface Weather Map data of `InputTracker`.

Ground heat flux (Q_g) is computed as

$$Q_g = 0.5Q^* \quad [\mathrm{Wm}^{-2}]. \tag{3.17}$$

3.5.2 Solving for Balance

Because each term is determined by the surface temperature T_s, no single term can be computed directly. Instead, we use the Newton-Raphson iterative method [34] to solve the equation. In this approach, for each iteration a guess is made for the value of T_s, allowing the balance equation and its derivative to be computed. These solutions are used to determine a the guess of T_s for the next iteration. With each iteration the balance equation will converge toward zero. Once its value is within some threshold, the current value of T_s is stored.

Note that the amount of longwave energy emitted by patches is dependent on T_s. This means *every iteration* of the Newton-Raphson method requires a new simulation of emitted longwave energy. All patches must re-emit longwave rays and accumulate energy at their surfaces emitted by other patches. To simplify this re-emission, atmospheric and solar radiation are stored in a separate buffer. After patch longwave

```
Function sphere_direction( ray_id, samples )
    offset = 2 / samples
    y = ray_id * offset - 1 + ( offset / 2 )
    r = sqrt( 1 - y*y )
    phi = ray_id * Pi * ( 3 - sqrt(5) )
    return normalize( [ cos( phi )*r, y, sin( phi )*r ] )
```

Figure 3.9: Pseudo-code for calculating the spherical ray direction using the modified spiral points method.

re-emission, the total accumulated energy at the patch is summed. Still, the simulation of terrestrial longwave exchange is a computationally expensive procedure. This is one of the driving motivations for requiring a highly efficient radiation model.

3.6 Sampling

An extremely important component of ray tracing is the method by which ray directions and origins are sampled. In `QESRadiant` and `QESViewfactor`, this exists in hemispherical and spherical emission of rays for radiation exchange and sky view factor calculations. Inadequate sampling may lead to aliasing or inaccurate exchange of energy within the domain.

3.6.1 Spherical Emission

When rays are scattered or emitted by vegetation volumes, they are launched in a sphere about its center. To compute ray direction, evenly distributed points of a unit sphere are sampled. The points on the sphere are sampled using a modification of the Spiral Points algorithm by A. B. J. Kuijlaars and E. B. Saff [39]. To obtain a more evenly distributed set of points, the Golden Ratio is introduced in the traversal of the spiral nodes.

A thread will call a function to compute this direction by passing in its ray id and the total number of rays per sphere. The ray id is an integer between 0 and the number of rays per sphere. This function is defined in figure 3.9.

3.6.2 Hemisphere Emission

The ray directions used to approximate sky view factors, wall view factors, and surface radiation scattering and emission, are computed by sampling a hemisphere about the center of a patch. The function defined in figure 3.9 can still be used. However, the total number of samples passed as an argument is *doubled* such that only half of the points on a sphere are computed. This hemisphere is then rotated to project in the direction of the patch normal.

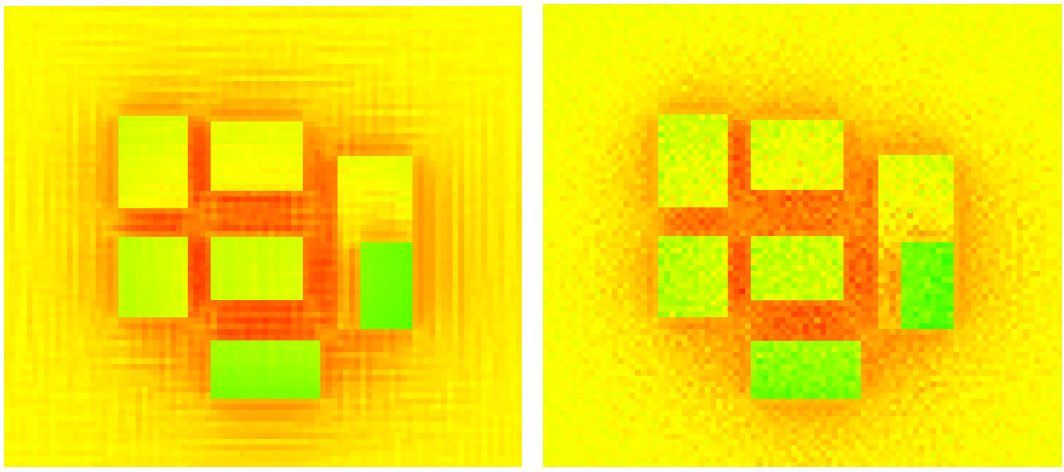
3.6.3 Random Rotations

After the sphere or hemisphere has been sampled, their direction is rotated by a random number between 0 and 2π. This random number is the same for each ray of the same sphere/hemisphere, but will differ from other sampled spheres/hemispheres. This reduces the effects of aliasing in which patterns may be introduced due to uniform sampling. This effect is illustrated in figure 3.10.

Note that incoming diffuse solar radiation takes the ray azimuth and zenith into account when computing the radiance distribution, $N(\theta_r, \Gamma_r)$. Thus, the rotation of the hemisphere must be applied before $N(\theta_r, \Gamma_r)$ is computed.

3.6.4 Ray Origins

For spherical ray launches from a vegetation volume, the ray origin is translated from the center of the volume in the direction of the ray. This origin is placed on



(a) Uniform sampling introduced patterns such as the grid above.

(b) Randomly rotating the hemisphere produces more even and realistic output.

Figure 3.10: A top-down view of a small urban scene showing absorbed longwave energy, with (b) and without (a) randomly rotating the longwave emission hemisphere.

the surface of the volume to avoid self intersection when the ray is launched. For hemispherical ray launches about a surface, the ray origin is simple placed in the center about the patch.

Sun view factor and direct solar radiation rays have origins that are distributed about the surface of a patch. They are then jittered, in which a small random value is added to the origin to reduce aliasing effects.

3.7 Acceleration Techniques

Recall that QES was designed to accelerate the modeling of the urban microclimate. `QESRadiant` and `QESViewfactor` take advantage of numerous performance guidelines and practices. This section will focus heavily on techniques that can be used in NVIDIA OptiX and GPGPU.

There are several implementation details that effect performance. These were discovered when developing models that used OptiX. In addition, the OptiX Pro-

gramming Guide contains many performance guidelines that were adhered to during implementation of QES [27]. Such guidelines include

- Iterative instead of recursive ray launches to minimize stack overhead.
- Shallow and efficient geometry groups.
- Floating point operations and avoidance of type promotion in arithmetic.
- Minimization of live state across `rtTrace` calls in launch kernels.
- Reuse of launch kernels with variables to modify launch parameters.

3.7.1 Callable Programs

A callable program is an OptiX program type allows the host to change the target of a function call after the kernel has been loaded by the context. It has the benefit of reducing initial compile time, as well as provide easy variability in kernel execution. Due to these benefits, their use is recommended by the OptiX Programming Guide. However, each time a callable program is set by the host, the context must recompile the kernel. If an application switches the callable program frequently, this will drastically impede performance due to regular kernel recompilation.

In `QESRadiant`, performance was gained by replacing callable programs with static inlined functions. This was after it was acknowledged that the callable programs were being set too frequently during the simulation. To adjust at run time which function was called, an OptiX variable and switch statement was used. This resulted in a slightly larger initial compilation time, but an overall significant improvement in performance.

3.7.2 Atomic Operations

Because of the potential for multiple rays to be simultaneously depositing energy into the same patch, vegetative volume, or sensor, the location in memory of the addition needs to be *locked* until the addition is complete. Once the lock is released, the other ray can then add energy to that object's location within device memory, called an **atomic addition**. This may cause excessive thread divergence as the ray samples per patch or vegetation get larger, because rays that travel in similar directions are likely to arrive at the same destination. This is illustrated in figure 3.11.

To reduce the thread divergence caused by waiting for lock release, input buffers can be scaled to handle more concurrent writes. The length of energy storage buffers are increased by a factor dependent on the number of ray samples. That is, each object contains multiple memory locations that energy can be deposited into. After the ray launch is complete, a separate kernel combines the energy absorbed in the multiple memory locations to one value. At the cost of GPU memory, this results in significantly fewer concurrent atomic writes. This is illustrated in figure 3.12.

Beyond a certain scaling factor, however, scaling ceases to have an effect on run

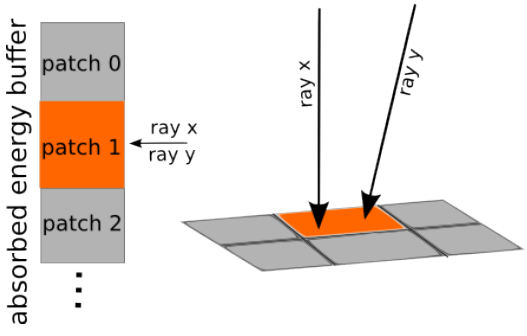


Figure 3.11: When two rays hit the same patch at the same time, one must wait for the other to finish writing to the buffer element of the patch.

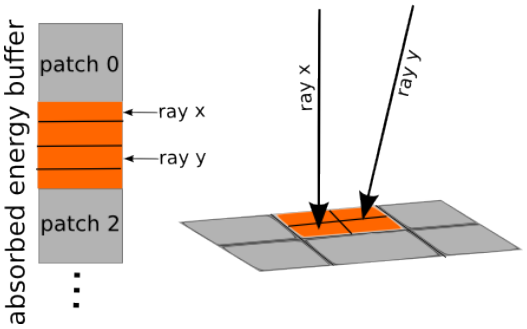


Figure 3.12: Increasing the amount of write locations per patch allows multiple rays to write to patch buffers.

time performance. This upper bound is dependent on the direction of the rays, the sample size N, and the simulation being executed. Thus, empirical tests must be run on a per-use basis to determine the appropriate scaling factor.

A caveat of atomic operations in OptiX is that they are not guaranteed in multi-GPU environments. Because `OUTPUT` and `INPUT_OUTPUT` buffers are stored on the host in this case, devices have no method of communicating lock information. In order to solve this problem, buffers must be stored on the device and post processed with CUDA. This is discussed in section 3.7.3.

3.7.3 Multiple GPUs

Recall that a major goal of QES is to scale the workload and utilize available hardware. However, many challenges impose the utilization of multiple GPUs. In OptiX, the copying of data between the host and device and tiling of threads on the GPU hardware is handled for you by the engine. This has several implications that further the difficulty of multi-GPU programing. This section will focus on optimizations specifically for multiple GPUs and OptiX.

Buffer Locality

One of the major implications of a multi-GPU environment is that `OUTPUT` and `INPUT_OUTPUT` buffers will default to host-local. This means the data resides in the host, and any read/write memory access from the device requires a host-device mapping. As seen in figure 3.14, this can be extremely detrimental to the efficiency of the simulation. In addition, GPUs currently have no way of communicating locks to one another. Because each GPU may write to the host buffer at the same time with an atomic operation, there is now a possibility of two GPUs writing to the same

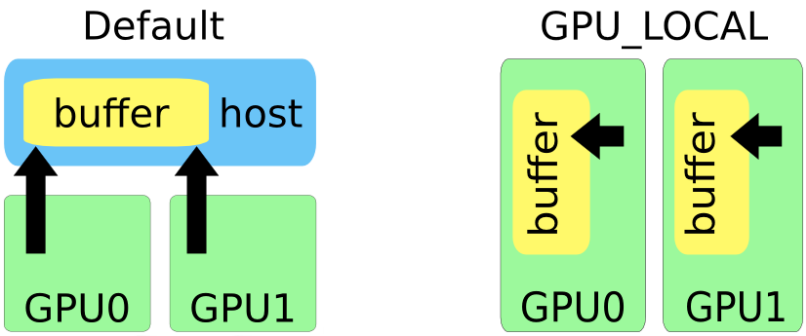


Figure 3.13: `OUTPUT` and `INPUT_OUTPUT` buffers reside on the host with default settings (left) requiring a device to host mapping every time a value is set. Flagging `INPUT_OUTPUT` with `GPU_LOCAL` forces them to reside on the device (right), drastically increasing write speeds. However, each device will retain a unique copy of the buffer and OptiX disables host-read access.

location. Fortunately, both of these problems can be solved by using `INPUT_OUTPUT` buffers and flagging them with `RT_BUFFER_GPU_LOCAL`. This tells the OptiX context that these buffers will reside on the device. However, this cannot be applied to `OUTPUT` buffers. Because the memory transactions will be device-local, atomic operations are now reliable, and there is no substantial cost to host-device mappings.

With device-local memory, each GPU maintains its own copy of the buffer. During the simulation each GPU will write to its respective buffer, resulting in two differing buffers with the same identifier. It's for this reason that the Optix context disables read access to any buffer flagged with `GPU_LOCAL`. However, the data is still obtainable through CUDA via `cudaMemcpyDeviceToHost`. To do this, a device pointer must be obtained from each GPU through the OptiX context. These operations are hidden from the user via the wrappers of `BufferTracker`. One function call exists to retrieve data from the device:

```
std::vector<type> buff_data;
qes::BufferTracker::getBuffer<type>( std::string buff_name, &↩
    buff_data );
```

In this line of code, the `BufferTracker` will inspect the requested buffer and perform necessary operations to retrieve data. In the case of `GPU_LOCAL` buffers, the data is cached on the host. If a kernel has not been launched and subsequent calls to `getBuffer` are issued, the cached data is returned, reducing the amount of device-to-host transactions.

Device Pointers

The most efficient way to access OptiX device data in CUDA is through the use of device pointers. The device pointer is an address that points to the first element of

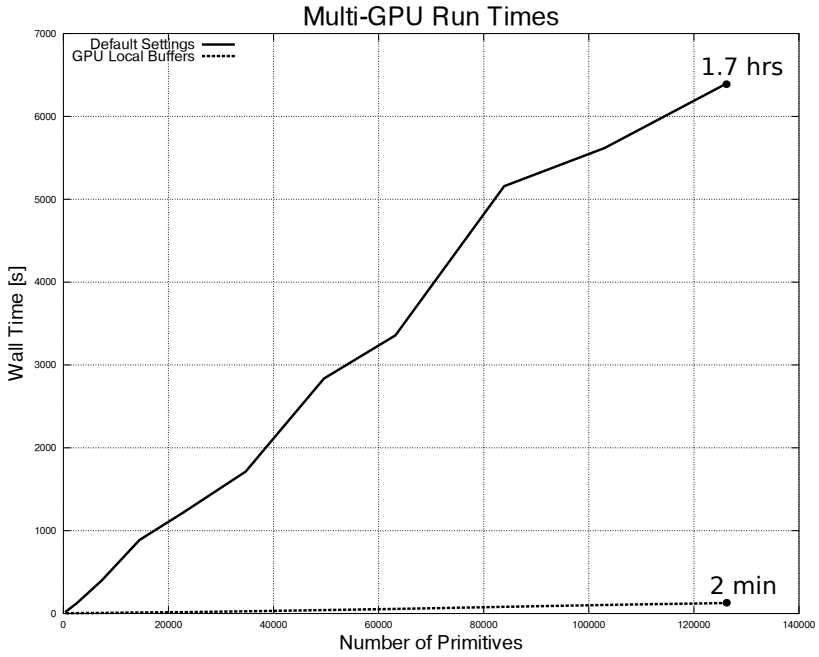


Figure 3.14: Multi-GPU in OptiX with default settings can be detrimental to application run time. This figure compares `QESRadiant` performance with default settings and `GPU_LOCAL` buffers.

an array in device memory, and must be obtained from the OptiX context. In OptiX 3.0 and later, calling `rtBufferGetDevicePointer` will create a CUDA context (if one has not already been created) and return a device pointer from a specified device. If the user does not call `rtBufferGetDevicePointer` for each GPU used by the system, OptiX will automatically copy the contents of the buffer for the device that *was* specified into the buffers of other devices on the next kernel launch.

However, it's worthwhile to note that the device ids returned from the OptiX context do not correlate exactly with the true device ids. So the originals must be stored and retrieved later when setting the compute device for CUDA or copying data from a particular GPU.

QES Automation

The `BufferTracker` provides an abstraction for creating and interacting with OptiX buffers. It essentially *wraps* the more technical details of setting and retrieving device data. As such, the `BufferTracker` must be able to handle the technical overhead of multiple GPUs, and no additional knowledge of multi-GPU requirements should be necessary. There are several steps that must be taken to carry out this task.

When a user creates a buffer through the `BufferTracker` the number of GPUs used by the system is queried. If this is larger than one and the buffer type is `RT_BUFFER_INPUT_OUTPUT`, then it will be flagged with `RT_BUFFER_GPU_LOCAL`. However, if a user decides they do not want this buffer to forced gpu-local, they can flag it as such. The name of the buffer is added to a list that stores all buffers flagged with `RT_BUFFER_GPU_LOCAL`. When retrieving the contents of a buffer from the `BufferTracker`, values are returned in an `std::vector`. So in a multi-gpu environment, the single vector will be filled with the contents of the device data from each

GPU. That is, for a device buffer that is N elements large, the size of the returned `vector` will be $N \times NumGPUs$. While this allows atomic functions and greatly reduces run time, it is not ideal for many applications. For example, `QESRadiant` records all incident radiation on an object. This requires the buffers from the multiple GPUs to be summed before the next kernel launch, such as computing the energy balance equation. This would require the host to obtain device data, sum its elements, and copy it back to each GPU.

Instead of handling it manually, this process is automated by the `BufferTracker`. Upon creation a buffer can be assigned a **join type**. The join type specifies how to combine the data from different GPUs, explained in section 3.7.3. For example, nearly all buffers needed for `QESRadiant` are assigned `QES_BUFFER_JOINTYPE_SUM`. Other options include `JOINTYPE_AVERAGE` and `JOINTYPE_PRODUCT`. Once a buffer is joined, it allows the `vector` returned by the `BufferTracker` to be N elements in length. By default there is no join type, defined `JOINTYPE_NONE`, and a host retrieval of the data will return $N \times NumGPUs$ elements. This allows all `INPUT_OUTPUT` buffers to reside on the device and not suffer from the extreme performance degradation of host residence.

When a buffer should be *joined* is another matter of consideration. In general, it's assumed that the contents of output buffers may be altered during kernel execution. Thus, an OptiX program launch will trigger a join phase by the `BufferTracker`. After kernel execution, any buffer that has a specified join type will be automatically joined. It will not be joined again until another kernel is launched.

Buffer Joining

Joining device buffers can be done without any device to host data transactions. There are four steps taken by the `BufferTracker`:

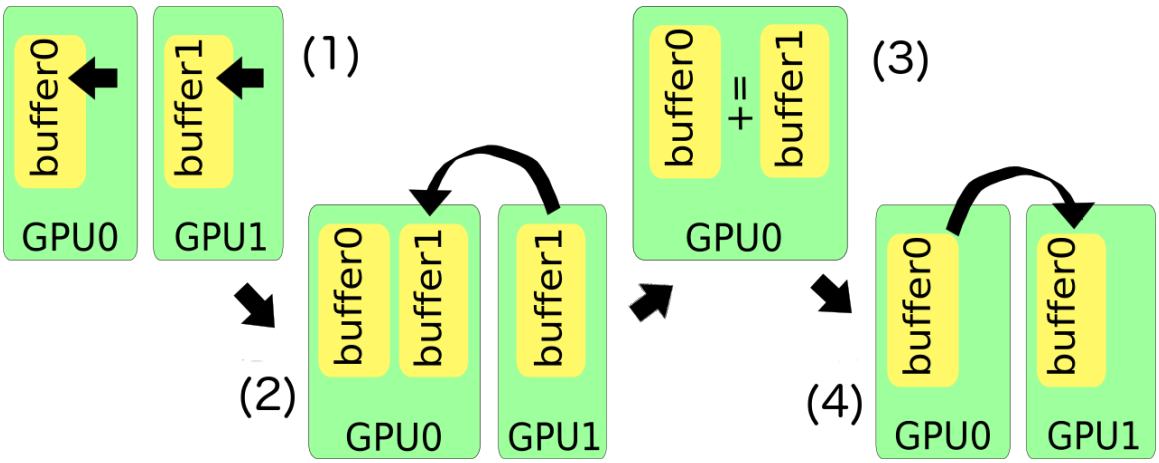


Figure 3.15: Automatic handling of `INPUT_OUTPUT` buffers in multi-GPU environments.

1. Flag buffers with `RT_BUFFER_GPU_LOCAL`.
2. Copy the contents of buffers from other GPUs to one device.
3. Join them into one element, e.g. sum, product, or average.
4. Copy the joined buffer back to other GPUs.

These four steps are illustrated by figure 3.15.

Step one is explained in 3.7.3. Step two can be completed by passing the device pointers of the OptiX buffers to a CUDA function and using `cudaMemcpyPeer`. A simplified example of how this is done with four-byte elements is presented below. In the following function, `buff_d0` is a device pointer to the data on device zero, and `buff_d1` is a device pointer to the buffer with the same identifier on device one.

```
void host_join( int n_elements, void *buff_d0, void *buff_d1 ){
   ...
   cudaSetDevice(0); \\ Do join on device 0
   float *gpu1_copy; \\ Need to copy device 1 data to device 0
   cudaMalloc( (void**) &gpu1_copy, 4*n_elements ); // Allocate space.
   cudaMemcpyPeer( gpu1_copy, 0, buff_d1, 1, 4*n_elements ); // Copy!
```

The joining of device buffers in step three is done with a CUDA kernel. While sum, product, and average are the currently the only join types available, there is nothing that restricts the user from adding additional join types. Note that unlike OptiX, buffers in CUDA can be both written to and read from in the same kernel. A buffer sum is shown in the following code:

```
__global__ void device_sum( int n_elements, float *buff_d0, float *↩
    buff_d1 ){
    ... // Get thread index and make sure it is less than n_elements
    buff_d0[index] = buff_d0[index] + buff_d1[index];
}
```

After the contents of the joined buffer is copied to other GPUs, additional copies are deallocated. A simplified host wrapper for the entire join operation is shown below for float buffers:

```
void host_join( int n_elements, void *buff_d0, void *buff_d1 ){
    ...
    cudaSetDevice( 0 ); \\ Do join on device 0
    float *gpu1_copy; \\ Need to copy device 1 data to device 0
    cudaMalloc( (void**) &gpu1_copy, 4*n_elements ); // Allocate space
    cudaMemcpyPeer( gpu1_copy, 0, buff_d1, 1, 4*n_elements ); // Copy
    device_sum<<< ... >>>( n_elements, (float*)buff_d0, gpu1_copy ); ↩
        // Sum
    cudaMemcpyPeer( buff_d1, 1, buff_d0, 0, 4*n_elements ); // ↩
        Redistribute
    cudaFree( gpu1_copy ); \\ Remember to deallocate copies!
}
```

Note that these code examples are for a single type (float) to simplify the illustration. The `BufferTracker` uses template functions and can manage buffers that are dynamically typed. However, the CUDA kernels used in a buffer join are *not* dynamically typed but do support a wide range of literals. Thus, any buffers with custom types (such as structs) are automatically assigned `QES_BUFFER_JOINTYPE_NONE`. Any

attempt to alter this will prompt warnings from the system.

3.7.4 Memory Layout

Improper data alignment can have negative effects on GPU kernel execution [28]. This can sometimes result in errors and runtime degradation. Typically in CPU central computing, byte aligning data structures involved grouping inner variables according to the largest element.

However, in GPU computing with CUDA, a single thread can only read up to 16 bytes of data from global DRAM. Thus, lookups achieve maximum efficiency when data structures are grouped in sets of 4, 8, or 16 bytes. Depending on compute capability, this data may, or may not be cached on the chip. QES requires a significant amount of global memory to store physical properties of materials, simulation input, and model output. Therefore it is necessary to optimize the way this data is read from and written to on the device without regard to compute capability.

Because of this, values that are associated with the same task are grouped together into multi-element data structures. For example, when a radiation ray intersects a patch, it must know its albedo for each wavelength (PAR, NIR, and LW), as well as its diffuse and specular reflection ratios. Thus, one element of four floats is used for this entire group of information. This 16 byte element contains the three albedos, and diffuse fraction, δ_d. The direct fraction can simply be computed as $1 - \delta_s$. Any model developed in QES has access to these data structures, and need not know about the performance benefits they provide.

4 Results

4.1 API Example

Recall the first objective of this research:

> *Design a dynamic, extensible framework for urban microclimate modeling.*

Section 4.1.1 shows a sample test case in which QES is used in conjunction to the `QESViewfactor` module. Section 4.1.2 decomposes the implementation details of this sample to express how this objective is accomplished. To illustrate how new models can be coupled with existing ones to provide future extensibility, section 4.1.3 expands this test case.

4.1.1 A Sample Test Case

As a hypothetical example, a researcher wants to compute sun view factors of a domain specified by an XML file. Because they aren't familiar with C++, they want this data to be sent to a file where they can manipulate it with MATLAB. This requires the researcher to develop a test case and interact with the `BufferTracker` to obtain output. The full source for this test case can be seen below:

```
#include "QESContext.h"
#include "ViewTracer.h"
int main( int argc, char** argv ) {
   qes::QESContext context; // Create a context
```

```
   qes::ViewTracer viewTracer; // Use the view tracer to compute view ↩
      factors
   context.joinModel( &viewTracer ); // and join it with the context
   // Now we need to create our scene with the SceneTracker.
   qes::SceneTracker *g_sceneTracker = context.getSceneTracker();
   g_sceneTracker->initScene( "MyDomain.xml" );
   // Initialize and compile the system.
   if( !context.initialize() ){ return 1; }
   // Compute view factors. Make sure there was no error.
   bool success = context.runSimulation();
   if( success ){ // Output results if everything is okay
      // Buffers can be retrieved from the BufferTracker.
      qes::BufferTracker *g_buffTracker = context.getBufferTracker();
      // Dump it to a text file
      g_buffTracker->outputToFile<float>( "patch_fsun", "viewfactors.↩
         txt" );
   } // end simulation success
   return 0; // exit. The QESContext destructor takes care of cleanup!
} // end main
```

4.1.2 Breaking Down the Test Case

To understand what is happening, lets say the researcher ran the simulation on a quad-GPU machine. In the following line of code, the `Context` is created:

```
   qes::QESContext context; // Create a context
```

This begins the SETUP phase. All of the tools of QES are created and initialized. Some of the default settings have been applied, but may be over written by models or the user. Seeing there are four GPUs available, it creates an OptiX context with all four. It may run into an error in doing so, and will attempt to fix it depending on the exception type. If it cannot, the program will quit and print why.

```
   qes::ViewTracer viewTracer; // Use the view tracer to compute SVF
   context.joinModel( &viewTracer ); // and join it with the context
```

This shares the QES tools with the view factor module, `QESViewfactor`. At this point, additional models and modules could be joined to the system.

```
g_sceneTracker->initScene( "MyDomain.xml" );
```

The first thing that happens is the `SceneTracker` will do a recursive search from current directory for a file of that name. If one isn't found, it will report an error. If it finds "MyDomain.xml", it parses the file with the high performance XML parser `pugixml` [20]. Buildings are inspected such that they are reasonable dimensions (e.g. non-negative), then discretized to patches. Each patch is assigned materials from the XML file. If not described in the file, they are assigned default values. A ground is created. Trees are formed by filling geometric shapes with vegetation volumes. Sensors are added. But, the next line of code is the most important in all of QES:

```
if( !context.initialize() ){ return 1; }
```

This invokes the INITIALIZATION phase of program execution. If there is an error during initialization, we want to exit the program. The `Context` will attempt to allocate resources requested by every model that has been joined with the context. It converts domain data created by the `SceneTracker` into efficient geometrical representations and passes them to OptiX. Because this is with four GPUs, all output buffers requested by `ViewTracer` are automatically assigned `GPU_LOCAL`. Kernel programs are loaded and organized by the `ProgramTracker`. At the end of INITIALIZATION, the `Context` double checks everything to ensure all necessary requirements for OptiX have been fulfilled, such as assigning thread stack size.

```
bool success = context.runSimulation();
```

This will progress to the SIMULATION phase. If necessary, options can be adjusted and `runSimulation` can be called again. The kernels created by `ViewTracer` are executed. Common run time errors are automatically handled by the `Context`, such as stack overflows, in which the stack size is increased and the kernel is re-executed. The work load is split to the four GPUs.

```
g_buffTracker->outputToFile<float>( "patch_fsun", "viewfactors.txt"↩
    );
```

At this point, the `patch_fsun` buffer is still resident on the GPU and data has not yet been copied back to the host. Knowing there are four GPUs, when the `BufferTracker` is asked to output the device data to a file the following will happen:

- Device pointers are obtained from each GPU.
- The contents of each buffer are copied to one GPU.
- The four buffers are summed into one buffer.
- The results are copied to the other devices.
- Copies on the device doing the summation are freed.
- The result is copied to the host, and outputted to a file.
- The result is cached on the host. If a subsequent call to retrieve this buffer is made and no kernel has been executed, the cached result is returned to avoid the expensive device-to-host transaction.

```
    return 0;
```

This will invoke the TERMINATION phase by calling the `Context` destructor. All CPU memory is deallocated, including system input from the `InputTracker`, scene data from the `SceneTracker`, and any cached data used by QES for acceleration. Any device data that is not cleaned up by OptiX, such as scratch buffers created for CUDA, are freed.

In this example, only the most basic functions are used. If applicable, the user can change the number of ray samples, the default GPU thread stack size, the maximum number of GPUs to use, and more. QUIC EnvSim is dynamic and programmable, in that users can design custom model implementations and modules that use the same resources as base modules like `QESRadiant` and `QESViewfactor`.

4.1.3 Extending the Test Case

The hypothetical researcher decides he only wants to output the sun view factor if the sun's altitude is positive. Because this will be a regular operation in future test cases, the researcher wants to develop a model implementation to handle this condition.

The following model is developed:

```
#include "ModelBase.h"
namespace qes {
    class SunUpModel : public ModelBase {
    public:
        // The initialization function is where buffers and variables
        // can be created. The context will call this when necessary.
        bool initialize( SharedResources sr ){
            // Make a copy of the SharedResources struct
            // so we can use it again later. It contains a few
            // pointers to the QES tools.
```

```
        g_shared = sr;
        return true;
    }
    // The runSimulation function does the check.
    // It outputs sun view factors if the sun is up.
    bool runSimulation(){
        // First check if the solar altitude is positive.
        float alt = g_shared.sunTracker->getAltitude();
        bool sunup = ( alt > 0.f );
        // If the sun is up, output the values computed
        // by the view factor model.
        if( sunup ){ g_shared.buffTracker->outputToFile<float>
            ( "patch_fsun", "viewfactors.txt" ); }
        return true;
    }
  private:
    SharedResources g_shared;
  }; // end class SunUpModel
} // end namespace qes
```

And the model is including in the simulation by amending the following lines:

```
  qes::ViewTracer viewTracer; // Use the view tracer to compute view ↩
      factors
  context.joinModel( &viewTracer ); // and join it with the context
  qes::SunUpModel sunUpModel; // Use our new model
  context.joinModel( &sunUpModel ); // and join it with the context
```

The original implementation no longer has to make a call to the `BufferTracker` in the `main` function. When a call is made to `runSimulation`, the view factor model will be ran and its output stored in the `patch_fsun` buffer. Then, the `runSimulation` function of the SunUpModel will be ran. The `Context` calls the `runSimulation` functions of models in the order they were joined. If the view factor model failed to run or was never joined with the context, the `BufferTracker::outputToFile` call will simply return false and alert the user with an error.

This represents a simplified form of one-way coupling, but illustrates how different models can communicate. Even though the not-so-sophisticated SunUpModel relies on the view factor model, it does not interfere with its operations in any way. This

modular development is paramount in developing large complex systems of many interoperating components.

4.2 Computational Efficiency

Recall the second objective of this research:

> *Develop and test system components to facilitate high performance computing techniques. To automate certain functionality and allow future researchers to utilize such techniques.*

Note that it is difficult to compare computational efficiency of QES to other urban modeling frameworks and systems. Others ([9, 24, 21]) do not report much (if any) detail on simulation run time or hardware portability. Thus, considerations of computational efficiency were focused on analyzing if the automated functions of QES appropriately utilize hardware resources.

4.2.1 Multiple GPU

QES must be scalable both in domain size and computability. Thus if additional hardware is present, multiple GPUs in particular, QES must be able to take advantage of it.

Device Utilization

A major consideration before adding additional GPU resources is the ability to utilize them. In order to achieve reasonable scaling to additional GPUs, the work load must be evenly distributed to each GPU. The OptiX programming guide does not provide details how this can be done, so experiments were conducted to determine

how to appropriately balance the work load. The experiments of this section were conducted on an NVIDIA Tesla S2050, described in appendix D.1.

In this experiment, an OptiX program was launched that simply recorded which GPU the entry point was launched from in an `INPUT_OUTPUT` buffer:

```
RT_PROGRAM void store_device_ids() {
    ...
    int device_id = (int)rti_internal_register::reg_device_id;
    if( device_id == 0 ){ result = make_int4(1,0,0,0); }
    else if( device_id == 1 ){ result = make_int4(0,1,0,0); }
    ...
    output[thread_id] = result;
}
```

The output buffer from each GPU was summed to compute the total number of entry points launched per GPU. An important detail is that each entry point does not necessarily control an entire thread. Multiple entry points may be ran on a single thread, which is a potential optimization provided by the OptiX engine. Thus, it is not correct to assume that full utilization of GPU hardware provides the best performance. The different launch settings performed are:

1. One dimensional launch:

```
rtContextLaunch1D( context, 0, n_entry_points );
```

2. Two dimensional launch, all entry points in X:

```
rtContextLaunch2D( context, 0, n_entry_points, 1 );
```

3. Two dimensional launch, all entry points in Y:

```
rtContextLaunch2D( context, 0, 1, n_entry_points );
```

4. Two dimensional launch, split between X and Y:

```
int sqrt_n = sqrt( n_entry_points ); // assumes perfect ↩
    square
rtContextLaunch2D( context, 0, sqrt_n, sqrt_n );
```

5. Three dimensional launch, all entry points in X:

```
rtContextLaunch2D( context, 0, n_entry_points, 1, 1 );
```

6. Three dimensional launch, all entry points in Y:

```
rtContextLaunch2D( context, 0, 1, n_entry_points, 1 );
```

The second parameter of the launch call, 0, is the entry point identifier of the *store_device_ids* program. The *n_entry_points* variable was adjusted to illustrate how the OptiX engine distributes the entry points work load. The output presented in this section is the number of entry points executed on each GPU, listed between parenthesis. For example, "(100, 200, 0, 0)" would indicate 100 entry points operated on device 0, 200 entry points on device 1, and 0 entry points on the others.

Setting the launch size to 65536 (128^2) resulted in:

```
1D Launch: ( 65536, 0, 0, 0 )
2D Launch (X): ( 65536, 0, 0, 0 )
2D Launch (Y): ( 16384, 16384, 16384, 16384 )
2D Launch (X,Y): ( 65536, 0, 0, 0 )
3D Launch (X): ( 65536, 0, 0, 0 )
3D Launch (Y): ( 16384, 16384, 16384, 16384 )
```

Increasing launch size to 99856 (316^2) rays received the following result:

```
1D Launch: ( 65536, 34320, 0, 0 )
2D Launch (X): ( 65536, 34320, 0, 0 )
2D Launch (Y): ( 26128, 24576, 24576, 24576 )
2D Launch (X,Y): ( 64720, 35136, 0, 0 )
3D Launch (X): ( 65536, 34320, 0, 0 )
3D Launch (Y): ( 26128, 24576, 24576, 24576 )
```

This shows entry points are launched in groups of 65536, and each group is operated on devices in succession. For any launch size of 65536 or less, the entire work load will be computed on the first device and *no work* will be given to other devices. Unless the launch size is a multiple of 65536 and the number of GPUs, entry points will not be evenly distributed to multiple GPUs. The exception being launches entirely in the Y dimension.

However, a completely even distribution of work load between multiple devices is not always desirable. The purpose of grouping threads to be operated on one device is to afford optimizations of thread locality versus the scene trace. That is, rays that are near each other geometrically can be placed on cores that are near each other physically. This grants the optimization of allowing multiple rays acting on the same data simultaneously. Placing the entire work load in the Y dimension is not of the same dimensionality as the problem. Thus, OptiX cannot perform appropriate ray tiling and such optimizations are no longer available. In order to determine appropriate distribution of work between the dimensions, empirical tests must be conducted for a specific problem size.

Scalability

Using GPU_LOCAL buffers, we are able to improve performance with additional GPU resources. To validate this claim, experiments were conducted with `QESRadiant`, which is currently the most GPU-intensive module of QES. Run times were calculated with increasing domain complexity on an NVIDIA Tesla S2050 (D.1) with varied number of GPUs. Instead of setting the latitude, longitude, date, and time, the solar zenith angle was forced to 45 degrees.

The domain consisted of a ground plane that varied in the horizontal dimensions, and five layers of vegetative volumes above it. With each iteration of the test, the horizontal dimensions increased by two meters, adding new patches and vegetative volumes. The total count of these energy emitting and intercepting objects with associated run times is shown in figure 4.1.

Note that when the number of rays being emitted is not sufficiently large, there will not be increased thread utilization. For example, suppose we have four GPUs and each can handle 1024 threads concurrently. If we are launching 256 threads, splitting 64 threads to each GPU may actually be *slower* than launching all 256 on one device. However, the number of rays being emitted for QESRadiant is rarely less than tens of thousands, so it is able to achieve performance gains. These gains for a Tesla S2050 are shown on table 4.2. The values represent an average speedup through the entire experiment of variable scene dimensions.

Further investigation on how to improve the speedup is necessary. Ideally, a linear performance increase is achievable. This may be due to improper launch dimensionality, or cost of automated buffer joining after kernel launch by the `BufferTracker`.

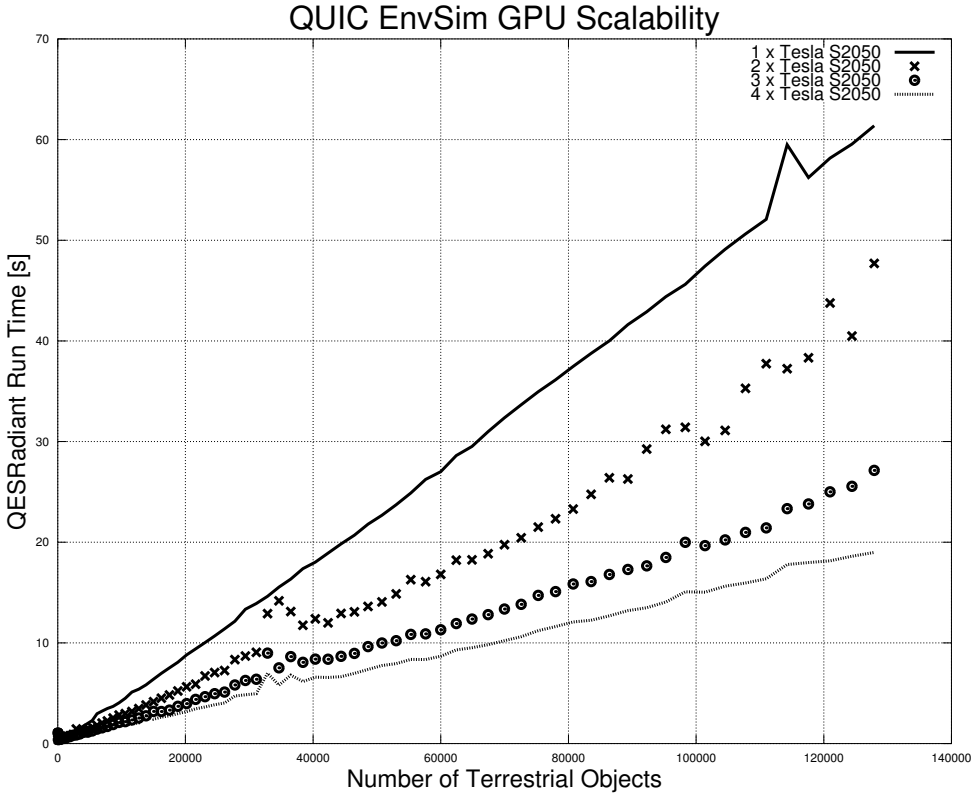


Figure 4.1: Performance of `QESRadiant` for 1-4 GPUs on a domain of increasing complexity.

	1 GPUs	2 GPUs	3 GPUs	4 GPUs
1 GPUs	-			
2 GPUs	1.45x	-		
3 GPUs	1.93x	1.31x	-	
4 GPUs	2.34x	1.57x	1.19x	-

Figure 4.2: The run time improvement of QESRadiant, represented as a multiplier averaged for all domain sizes.

4.2.2 Hardware Limits

It has been stated that QES is designed to handle large and complex domains. In the following test, the same test case as section 4.2.1 used, in which a scene expanded in size with each iteration. However, the scene expanded by 10 meters in the horizontal

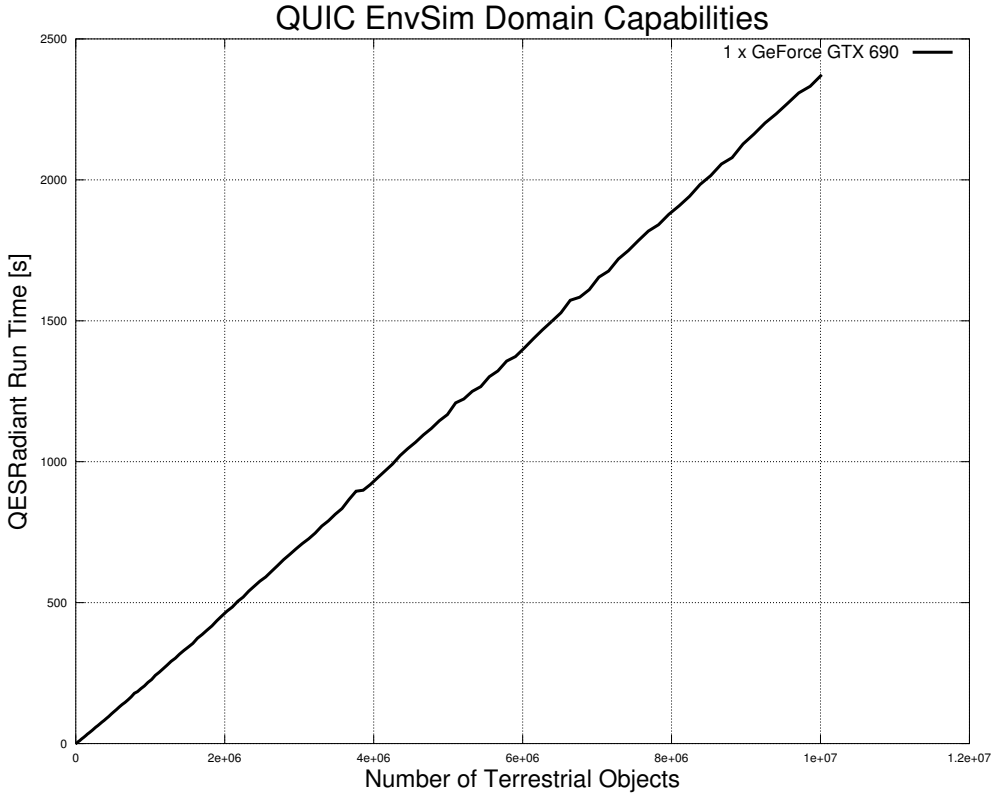


Figure 4.3: Performance of `QESRadiant` on a single GPU for a domain of increasing complexity.

directions. Once there was an error due to an OptiX exception (e.g. exceeding available memory), or inaccuracies in simulation output, the program exited. Run times recorded along with domain size. The experiment was ran on a single GeForce GTX 690, described in appendix D.1. Only `QESRadiant` was simulated, which is the most memory intensive module of QES.

`QESRadiant` was able to run a scene containing 1,669,264 patches and 8,346,320 vegetative volumes in 39.5 minutes. This does not include OptiX compile time or scene construction. The increase in computation run time to scene dimensions is almost perfectly linear.

4.3 Model Validation

Recall the third objective of this research:

> *Implement several important models for urban microclimate, including radiation transport, surface view factor, and a land surface model. Illustrate how models can be coupled, ablated, and validated.*

The radiation model and its coupled models (solar flux models, diffuse longwave flux, etc...) have been validated in related publications and presentations [2][29][30]. Validation and tests regarding the view factor approximations have also been conducted [15]. Additional sources for each model and submodel are available in chapter 3 where implementation details are described.

However, it is still important to illustrate how a model can be validated in QES. To ensure the underlying mechanisms of QES are operational, unit tests have been developed for several test cases. These tests are executed when new operations, models, or code is added to the framework. They will compare simulated output to measured data using statistics tools available in QES. Errors or deviance from past validations will produce error messages. Section 4.3.1 is an example of such validation for `QESRadiant`.

4.3.1 QESRadiant: MATERHORN Playa

The Mountain Terrain Atmospheric Modeling and Observations (MATERHORN) program is a multi-institutional, ongoing collection of studies on the atmosphere [12]. The main goals of the program are to bring together scientists of different fields and investigate the limitations of current climate models and produce knowledge to improve scientific modeling. To do so, there are several sites of varying landscapes, in which climate data is recorded over a period of time.

Measurement	R^2
Emitted Longwave Flux	0.967476
Reflected Shortwave Flux	0.988397

Figure 4.4: Coefficient of determination (R^2) for `QESRadiant`'s emitted and reflected fluxes.

The MATERHORN Playa site is located in the salt flats of Salt Lake City, Utah. In this experiment, incoming and outgoing radiation, as well as surface and air temperatures were recorded over several days. This experiment represents a *near-ideal* test case due to its simplicity. This means the models of `QESRadiant` should perform with near-perfect accuracy. The site contains no trees, buildings, or man-made materials. Because of its simplicity, the data obtained from the experiment can be used to validate the radiation model. Without the interaction of vegetation and buildings, as well as the low moisture content of the site, we should expect to see highly agreeable output.

The domain is modeled using the `smallflat` scene, which is described in figure C.1. Shortwave surface albedo was calculated from the first three days of measured data, and remained constant throughout the simulation. This value was assigned to the patches of the ground. Emissivity was set to 0.96. A downward facing sensor was placed 10 centimeters above the ground to record reflected and emitted radiation. The simulations that record reflected fluxes used measured values for unobstructed incoming fluxes from the atmosphere and sun as input.

These results can be seen in figures 4.5, 4.6, 4.7, and 4.8. Figure 4.4 provides a table of shortwave and longwave correlation in which the coefficient of determination (R^2) was computed using tools provided by `QESStatistics`. These values show a high correlation in which R^2 is close 1, indicating good agreement.

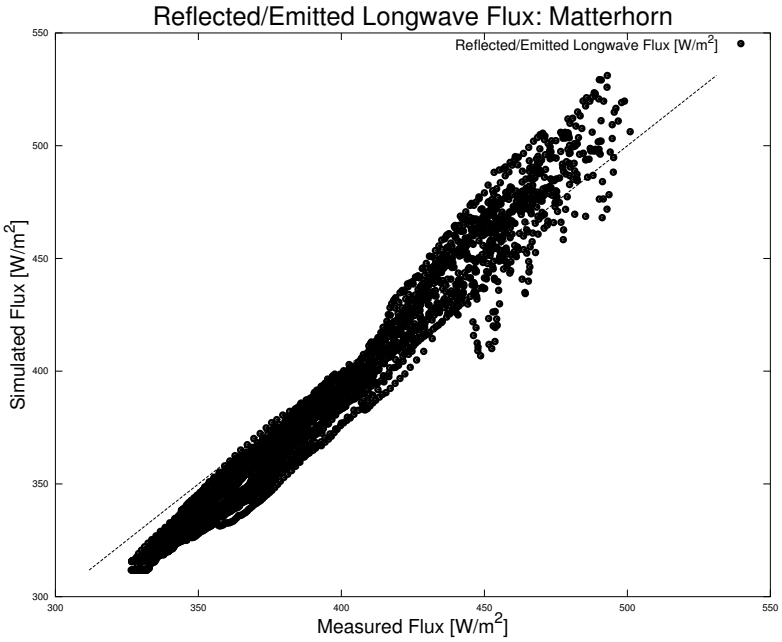


Figure 4.5: Emitted longwave flux from the MATERHORN Playa data set

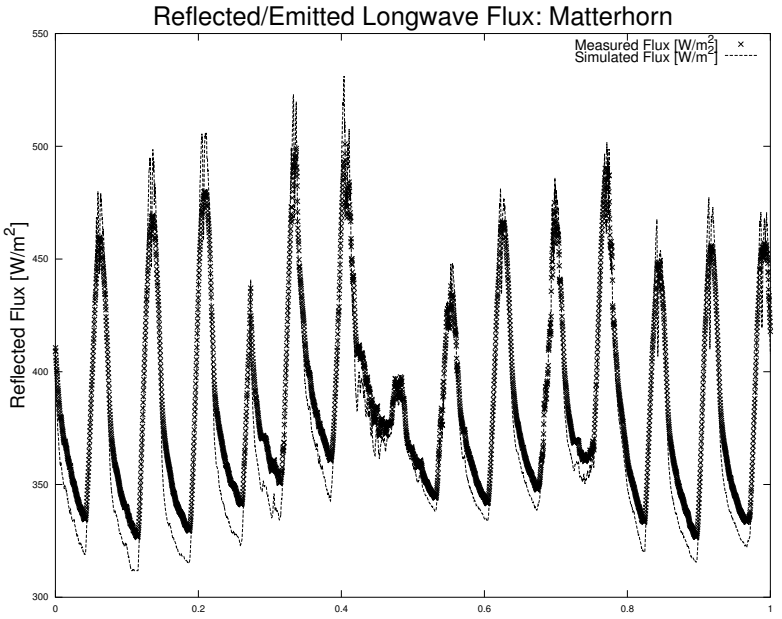


Figure 4.6: Emitted longwave flux from the MATERHORN Playa data set with respect to time

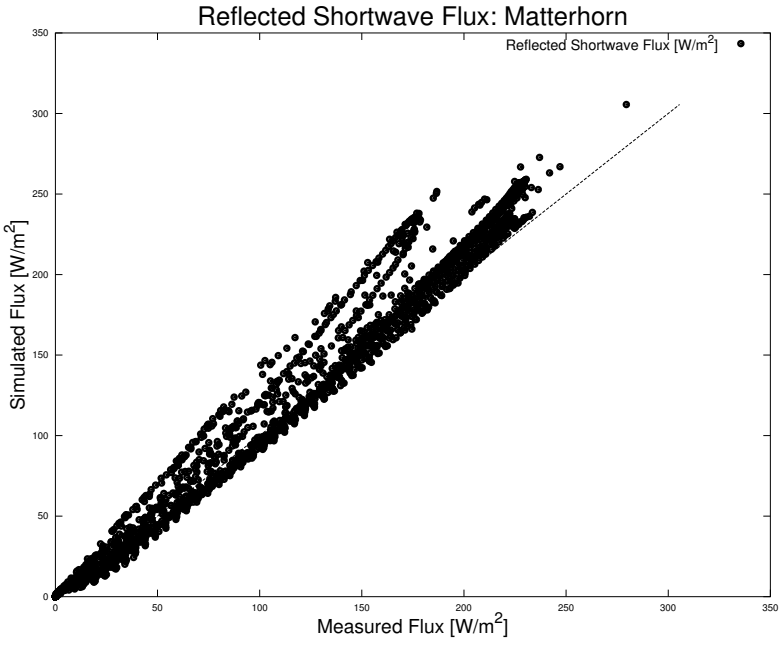


Figure 4.7: Reflected shortwave flux from the MATERHORN Playa data set

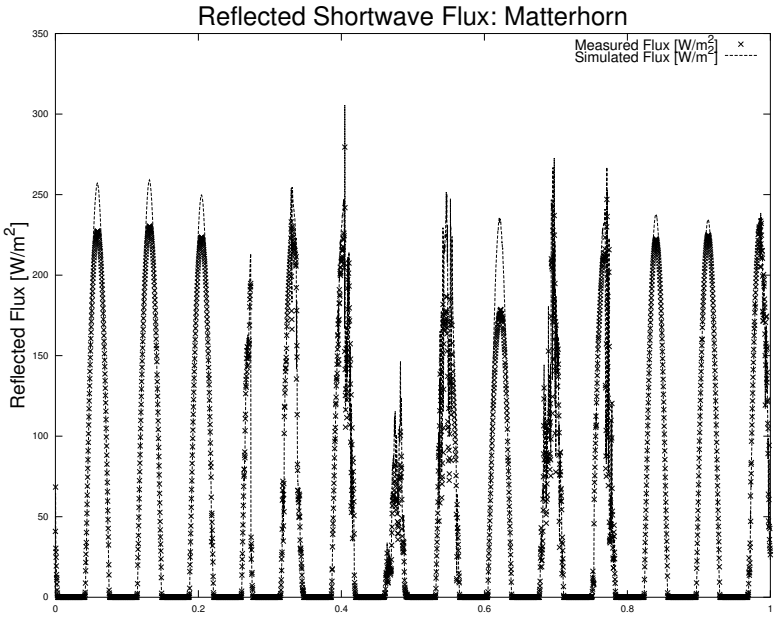


Figure 4.8: Reflected shortwave flux from the MATERHORN Playa data set with respect to time

5 Conclusions

There is a need for dynamic urban microclimate models that utilize high performance techniques for computational efficiency. To continue providing accurate output, these models must be able to be modified, ablated, and coupled with other models. Current high performance techniques are challenging, and present a high barrier for researchers utilizing such techniques in addition to providing model accuracy and viability.

This paper introduces QUIC EnvSim (QES), a dynamic, high performance framework for coupled urban microclimate models. Models can be developed for this framework for a wide range of aspects of the urban microclimate. Tools facilitate input and output communications between models and automate the process utilizing resources by scaling work load with additional computing hardware. In addition, many of the common requirements for microclimate modeling are handled efficiently by the system, such as domain resolution and standardized input.

QES incorporates GPU accelerated programming with CUDA and OptiX. Implementations of several sophisticated microclimate models have implemented and validated. `QESViewfactor` can approximate sun, sky, and wall view factors. `QESRadiant` can approximate radiation transfer in the forms of direct and diffuse solar, diffuse longwave, and reflected, scattered, and emitted longwave from building surfaces and vegetation. `QESLSM` solves a simplified energy balance equation and approximating surface temperatures. `QESGUI` provides interactive visualizations of system output. Several test cases and applications are provided to show the diverse and dynamic

capabilities of QES.

QES can handle large, complex urban domains. `QESRadiant` will operate on domains with several million discrete surfaces and vegetative volumes on the order of minutes. The research provided in this paper accomplishes the following goals:

1. Design a dynamic, extensible framework for urban microclimate modeling.

2. Develop and test system components to facilitate high performance computing techniques. To automate certain functionality and allow future researchers to utilize such techniques.

3. Implement several important models for urban microclimate, including radiation transport, surface view factor, and a land surface model. Illustrate how models can be coupled, ablated, and validated.

6 Bibliography

[1] G. Asrar, R. B. Myneni, and B. J. Choudhury. Spatial heterogeneity in vegetation canopies and remote sensing of absorbed photosynthetically active radiation: a modeling study. *Remote Sensing of Environment*, 41(2):85–103, 1992.

[2] Brian Bailey, Matthew Overby, Pete Willemsen, Eric Pardyjak, W. Mahaffee, and Rob Stoll. A scalable plant-resolving radiaitive transfer model based on gpu ray tracing. *Agricultural and Forest Meteorology*, 2014.

[3] Manuel Blanco-Muriel, Diego C Alarcón-Padilla, Teodoro López-Moratalla, and Martín Lara-Coira. Computing the solar vector. *Solar Energy*, 70(5):431–441, 2001.

[4] Wendell J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[5] Julien Bouyer, Marjorie Musy, Yuan Huang, and Khaled Athamena. Mitigating urban heat island effect by urban design: forms and materials. In *Proceedings of the 5th urban research symposium, cities and climate change: responding to an urgent agenda, Marseille*, pages 28–30, 2009.

[6] Kevin Briggs, M. Overby, D. Alexander, R. Stoll, P. Willemsen, and E. Pardyjak. Evaluation of moisture and heat transport in the building-resolving urban transport code quic envisim. AMS 2014, Symposium on the Urban Environment, 2014.

[7] Michael Brown. Quic-gui city builder, 2014.

[8] Michael J. Brown. Urban dispersion - challenges for fast response modeling. *Los Alamos National Laboratory*, 5129, 2004.

[9] Michael Bruse. Envi-met 3.0: updated model overview. *University of Bochum*, 2004.

[10] Bing Chen, D Clark, John Maloney, W Mei, and John Kasher. Measurement of night sky emissivity in determining radiant cooling from cool storage roofs and roof ponds. In *Proceedings of the National Passive Solar Conference*, volume 20, pages 310–313. American Solar Energy Society, 1995.

[11] Joshua Clark. A fast and efficient simulation framework for modeling heat transport. Master's thesis, University of Minnesota Duluth, 2012.

[12] H. Fernando, E. Pardyjak, D. Zajic, S. De Wekker, and J. Pace. The mountain terrain atmospheric modeling and observations (materhorn) program: The first field experiment (materhorn-x1). In *AGU Fall Meeting Abstracts*, volume 1, page 01, 2012.

[13] Lewis Gill, E. Abigail Hathway, Eckart Lange, Ed Morgan, and Daniela Romano. Coupling real-time 3d landscape models with microclimate simulations. *International Journal of E-Planning Research*, 2(1):1–19, 2013.

[14] Renate Hagedorn, Francisco Doblas-Reyes, and T.N. Palmer. The rationale behind the success of multi-model ensembles in seasonal forecasting - i. basic concept. *Tellus A*, 57(3):219–233, 2005.

[15] Scot Halverson. Energy transfer ray tracing with optix. Master's thesis, University of Minnesota Duluth, 2012.

[16] A. W. Harrison and C. A. Coombes. Angular distribution of clear sky short wavelength radiance. *Solar Energy*, 40(1):57–63, 1988.

[17] R. G. Hintz and D Tate. Control data star-100 processor design. In *Compcon*, 1972.

[18] John Horel, Michael Splitt, L. Dunn, J. Pechmann, B. White, C. Ciliberti, S. Lazarus, J. Slemmer, D. Zaff, and J. Burks. Mesowest: Cooperative mesonets in the western united states. *Bulletin of the American Meteorological Society*, 83(2):211–225, 2002.

[19] Hamlyn G. Jones and Robin A. Vaughan. *Remote sensing of vegetation: principles, techniques, and applications*. Oxford university press, 2010.

[20] Arseny Kapoulkine. Pugixml, 2014.

[21] E. Scott Krayenhoff and James A. Voogt. A microscale three-dimensional urban energy balance model for studying surface temperatures. *Boundary-Layer Meteorology*, 123(3):433–461, 2007.

[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[23] Benjamin Y. H. Liu and Richard C. Jordan. The interrelationship and characteristic distribution of direct, diffuse and total solar radiation. *Solar Energy*, 4(3):1–19, 1960.

[24] Andreas Matzarakis, Frank Rutz, and Helmut Mayer. Modelling radiation fluxes in simple and complex environmentsapplication of the rayman model. *International Journal of Biometeorology*, 51(4):323–334, 2007.

[25] Michael Mishchenko, Larry Travis, and Andrew Lacis. *Scattering, absorption, and emission of light by small particles*. Cambridge University Press, 2002.

[26] John Monteith and Mike Unsworth. *Principles of Environmental Physics: Plants, Animals, and the Atmosphere*. Academic Press, 2008.

[27] NVIDIA. Nvidia optix ray tracing engine programming guide. Version 3.0, 2012.

[28] CUDA Nvidia. Nvidia cuda c programming guide. *NVIDIA Corporation*, 120, 2011.

[29] Matthew Overby, B. Bailey, R. Stoll, P. Willemsen, and E. Pardyjak. Simulating radiative transport for vegetation in complex urban environments with green infrastructure. AMS 2014, Symposium on the Urban Environment, 2014.

[30] Matthew Overby, Brian Bailey, Rob Stoll, Peter Willemsen, and Eric Pardyjak. A highly scalable modeling framework based on gpu technology for simulating radiative transport in complex urban and plant canopies. ESA 2013, Sustainability: Urban Systems, 2013.

[31] Matthew Overby, Scot Halverson, Brian Bailey, Pete Willemsen, Rob Stoll, and Eric Pardyjak. Quic envsim: Radiative heat transfer in vegetative and urban environments with nvidia optix. GPU Technology Conference 2014, 2014.

[32] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[33] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. In *ACM Transactions on Graphics*, volume 29, page 66. ACM, 2010.

[34] Zhihao Qin, Pedro Berliner, and Arnon Karnieli. Numerical solution of a complete surface energy balance model for simulation of heat fluxes and surface temperature under bare soil environment. *Applied mathematics and computation*, 130(1):171–200, 2002.

[35] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE micro*, 20(4):47–57, 2000.

[36] D. Robinson, N. Campbell, W. Gaiser, K. Kabel, A. Le-Mouel, N. Morel, J. Page, S. Stankovic, and A. Stone. Suntool - a new modelling paradigm for simulating and optimising urban sustainability. *Solar Energy*, 81(9):1196–1211, 2007.

[37] J. Ross. *The radiation regime and architecture of plant stands*. Springer, 1981.

[38] F. B. Rowley and W. A. Eckley. Surface coefficients as affected by wind direction. *ASHRAE Trans*, 38:33–46, 1932.

[39] Edward B. Saff and A. B. J. Kuijlaars. Distributing many points on a sphere. *The Mathematical Intelligencer*, 19(1):5–11, 1997.

[40] Mat Santamouris. *Environmental design of urban buildings: an integrated approach*. Routledge, 2013.

[41] Matheos Santamouris. *Energy and climate in the urban built environment*. Routledge, 2013.

[42] Brian Smits. Efficiency issues for ray tracing. In *ACM SIGGRAPH 2005 Courses*, page 6. ACM, 2005.

[43] C. J. T. Spitters. Separating the diffuse and direct component of global radiation and its implications for modeling canopy photosynthesis part ii. calculation of canopy photosynthesis. *Agricultural and Forest meteorology*, 38(1):231–242, 1986.

[44] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[45] D. F. Swinehart. The beer-lambert law. *Journal of chemical education*, 39(7):333, 1962.

[46] Werner H. Terjung and Patricia A. O'Rourke. Simulating the causal elements of urban heat islands. *Boundary-Layer Meteorology*, 19(1):93–118, 1980.

[47] Xinyan Yang and Yuguo Li. Development of a three-dimensional urban energy model for predicting and understanding surface temperature distribution. *Boundary-layer meteorology*, 149(2):303–321, 2013.

[48] Zhiyi Yang, Yating Zhu, and Yong Pu. Parallel image processing based on cuda. In *International Conference on Computer Science and Software Engineering*, volume 3, pages 198–201. IEEE, 2008.

A QES Additional Details

A.1 Urban Surface Materials

Shortwave albedos and material emissivities (ϵ) are reported by Matheos Santamouris [40]. Because the text did not differentiate between spectral bands, the average albedo was used for both PAR (α_{par}) and NIR (α_{nir}). Longwave albedo (α_ℓ) is assumed zero. Note that all physical properties are subject to change, and the user may add additional materials as needed.

Diffuse δ_d and specular δ_s fraction does not exist in literature, thus it was estimated. Future work may included defining more accurate terms for δ_d and δ_s.

Material	α_ℓ	α_{par}	α_{nir}	δ_s	δ_d	ϵ
Soil	0.0	0.17	0.17	0.0	1.0	0.94
Red Brick	0.0	0.3	0.3	0.0	1.0	0.9
Concrete	0.0	0.3	0.3	0.0	1.0	0.94
Glass	0.0	0.8	0.8	0.9	0.1	0.95
Wood	0.0	0.4	0.4	0.0	1.0	0.9
Gravel	0.0	0.72	0.72	0.0	1.0	0.28
Sand	0.0	0.24	0.24	0.0	1.0	0.76
Grass	0.0	0.21	0.21	0.0	1.0	0.93
White Paint	0.0	0.85	0.85	0.9	0.1	0.96
Tar Paper	0.0	0.05	0.05	0.0	1.0	0.93
Black Body	0.0	0.0	0.0	-	-	1.0

Figure A.1: Surface material properties as represented in QES.

B Source Code Examples

B.1 Example Test Case

The following example shows how to create an instance of QES, load a scene, initialize models, run a simulation, and exit the program.

```
#include "QESContext.h"
#include "RadiationTracer.h"
#include "QESGui.h"

int main( int argc, char** argv ) {

    // The first thing we need to do is create our QUIC EnvSim context.
    // The context controls many of the environment variables we need
    // for the system to run smoothly. It also acts as a wrapper
    // for OptiX and CUDA.
    //
    // We set it up by joining all the models we want, building the
    // scene, then initializing the context. Initialization of the ↩
        context
    // will compile everything together. This means no new geometry or
    // models can be added, and attempts to do so will be blocked.
    //
    // When running different model sets and different scenes within
    // the same executable, it is best to just destroy the QESContext
    // and build a new one. We pass the maximum number of GPUs
    // it is allowed to use as an argument.
    qes::QESContext context( 1 );

    // Since this is a sample, we might as well ↩
        check_energy_conservation
    // to true. This option will create additional GPU memory and keep
    // track of where ALL the energy ends up. This increases run time ↩
        and
    // memory usage, but can be very useful when debugging a new model.
    qes::VariableTracker *g_varTracker = context.getVariableTracker();
    g_varTracker->setBool( "check_energy_conservation", true );
```

```
// Now we create the RadiationModel and join it with the context.
// This allows it access to shared tools and provides a
// channel of communication with the system.
qes::RadiationTracer radModel;
context.joinModel( &radModel );

// Now we need to create our scene. It does not matter when this ↩
    gets
// done, as long as it is done before we initialize the QESContext.
// To do this, we need to get the context's SceneTracker.
qes::SceneTracker *g_sceneTracker = context.getSceneTracker();

// Lets load one of the sample scenes. The SceneTracker comes
// with a few scenes hard coded into the system, used for debugging↩
    .
// Urban Vege is my favorite because it has a good mix of ↩
    vegetation
// and urban form.
// Also, it totally shows off the glaring errors of the radiation
// model (boundary conditions and low sun altitudes).
g_sceneTracker->initScene( Testcase::urbanvege );

// Now that we've added all the models we want to use and created
// our scene, it's time to initialize and compile the system. This
// will return false if there is a problem.
if( !context.initialize() ){ return 0; }

// Before we run our simulation we should set it to a time where
// the radiation model isn't full of errors. Let's do 2pm
// on October 1st, 2014.
int hour = 14; int minute = 0; int second = 0;
qes::SunTracker *g_sunTracker = context.getSunTracker();
g_sunTracker->setTimeLocal( hour, minute, second );
g_sunTracker->setDate( 2014, 10, 1 );

// Now we can run the simulation. This can be done by calling
// QESContext::runSimulation, RadiationModel::runSimulation,
// or handling the submodels ourself.
context.runSimulation();

// We can create an interactive GUI to visualize the result.
// It also provides options for changing settings like latitude,
// longitude, date, and time. The simulation can be ran with
// helpful buttons and on-screen documentation.
int screen_width = 1024;
int screen_height = 768;
qes::QESGui gui( screen_height, screen_width, &context );
gui.display();

// Exit the program. The QESContext takes care of all the cleanup!
```

```
    return EXIT_SUCCESS;

} // end main
```

B.2 Manual Specification of a Scene

There are three factory classes to facilitate the construction of trees, buildings, and aircells. A factory class is one that does not need to be stored as an object in memory. Given input, it produces output. These factory classes also define parameters that are required by the system to define certain worldly objects, such as a surface's albedo or a tree's leaf area density. If not specified by the user as input, these parameters are set to defaults. A `BuildingBuilder` creates the ground and buildings and discretized them into patches. A `TreeBuilder` generates vegetation volumes from inputs classifying a tree. An `AircellBuilder` fills the space (outside of buildings and trees) with discretized volumes of air.

```
// The loadScene function takes in a QESContext and returns true if
// the scene was built correctly. The sample scene is a ground with
// a building and tree in opposite corners.
bool loadScene( qes::QESContext *context ){

    // Get QES tools and data
    qes::SceneTracker *g_sceneTracker = context->getSceneTracker();
    PatchMap *g_patchData = g_sceneTracker->getPatchData();
    BuildingMap *g_buildingData = g_sceneTracker->getBuildingData();
    TreeMap *g_treeData = g_sceneTracker->getTreeData();
    VegeMap *g_vegeData = g_sceneTracker->getVegeData();

    // Set scene parameters (in meters)
    float3 patchDim = make_float3( 1.f, 1.f, 1.f );
    float3 vegeDim = make_float3( 1.f, 1.f, 1.f );
    float3 worldDim = make_float3( 100.f, 100.f, 50.f );
    g_sceneTracker->setSceneParameters( patchDim, worldDim, vegeDim ↩
        );

    // Create our tree using the TreeBuilder factory class
```

```
    // Tree trunks are stored as buildings with wood physical ↩
       properties
    std::vector< BuildingData > tree_trunks;
    TreeBuilder treeBuilder( &g_vegeDatas, &g_treeDatas, vegeDim,
       patchDim, &tree_trunks );

    // Create the TreeData struct which specifies tree parameters
    TreeData newTree;
    newTree.trunkDiameter = 1.f;
    newTree.setShape( "cone" );
    newTree.position = make_float3( 90.f, 90.f, 0.f );
    newTree.crownRadius = 5.f;
    newTree.crownHeight = 10.f;
    newTree.height = 13.f; // 3 meter trunk
    treeBuilder.buildTree( newTree );

    // Use the BuildingBuilder factory class
    qes::BuildingBuilder buildingBuilder( g_patchData, ↩
       g_buildingData,
       patchDim, worldDim );

    // Add a building to the corner of the scene
    // Since it is not specified, the roof will default
    // to tar paper and the walls will be red brick.
    float3 boxmin = make_float3( 2.f, 2.f, 0.f );
    float3 boxmax = make_float3( 20.f, 20.f, 15.f );
    BuildingData newBuilding( boxmin, boxmax );
    buildingBuilder.buildBuilding( newBuilding );

    // Add the tree trunks
    for( int i=0; i<tree_trunks.size(); ++i ){
       buildingBuilder.buildBuilding( tree_trunks[i] );
    }

    // Finalize build
    if( !g_sceneTracker->checkSuccess() ){ return false; }
    return true;

} // end load scene
```

B.3 XML Scene File

The following XML text contains everything necessary to define a scene in QES. This simple domain is a 100 by 100 meter domain. It has a single rocket-shaped tree in one corner, and a small building in the other.

```
<?xml version="1.0"?>

<!-- Required scene variables -->
<projectName>Duluth Sample</projectName>
<patchDim>1 1 1</patchDim>
<vegeDim>1 1 1</vegeDim>
<worldDim>100 100 20</worldDim>
<latitude>46.8</latitude>
<longitude>-92.1</longitude>

<!-- Explicitly defined rocket-shaped tree -->
<treelist>
   <tree>
      <shape>Rocket</shape>
      <position>88 72 0</position>
      <height>10</height>
      <crownHeight>8</crownHeight>
      <crownRadius>4</crownRadius>
      <trunkDiameter>1</trunkDiameter>
      <coneHeight>1</coneHeight>
   </tree>
</treelist>

<!-- Simple building with default materials -->
<buildingList>
   <building>
      <boxmin>2 2 0</boxmin>
      <boxmax>20 20 15</boxmax>
   </building>
</buildingList>

<!-- One downward facing sensor -->
<sensorList>
   <sensor>
      <center>50 50 2</center>
      <normal>0 0 -1</normal>
   </sensor>
</sensorList>
```

B.4 Surface Weather Map XML File

A surface weather map (SWM) XML file generated by MesoWest [18] contains a list of **observations** at a given date, time, and geographic location. The following XML text contains a single observation. A user has the option of creating their own SWM from a different source, so long as the names of the observation parameters are consistent. Though the following example contains only one observation for a data set (`mesowest_data`), a single SWM file may contain any number of data sets and observations.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mesowest_data version="1.0" xmlns:xsd="http://www.w3.org/2001/↩
   XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <credit>MesoWest at the University of Utah</credit>
  <credit_URL>"http://mesowest.utah.edu"</credit_URL>
  <location>Salt Lake County, UT, US</location>
  <station_id>WRHU1</station_id>
  <station_name>WRH FEDERAL BUILDING IN SALT LAKE CITY</station_name↩
     >
  <latitude>40.7667</latitude>
  <longitude>-111.8867</longitude>
  <elevation>4300</elevation>
  <elevation_m>1310.64</elevation_m>
     <observation>
        <observation_time>Updated on May 31 2014, 0:00 GMT </↩
           observation_time>
        <temp_c>28.2</temp_c>
        <wet_bulb_c>28.17</wet_bulb_c>
        <water_temp_c>-17.8</water_temp_c>
        <relative_humidity>18.0</relative_humidity>
        <wind_dir>NW</wind_dir>
        <wind_degrees>304</wind_degrees>
        <wind_kt>5</wind_kt>
        <pressure_mb>0.00</pressure_mb>
        <altimeter_mb>0.00</altimeter_mb>
```

```
        <pressure_1500m_mb>0.00</pressure_1500m_mb>
        <pressure_sealevel_mb>-9999.00</pressure_sealevel_mb>
        <dewpoint_c>1.59</dewpoint_c>
        <heat_index_c>26.81</heat_index_c>
        <windchill_c>30.37</windchill_c>
        <visibility_km>0.00</visibility_km>
        <solar_radiation_w_m2>0.0</solar_radiation_w_m2>
    </observation>
</mesowest_data>
```

C Test Cases

C.1 Hard-Coded Test Cases

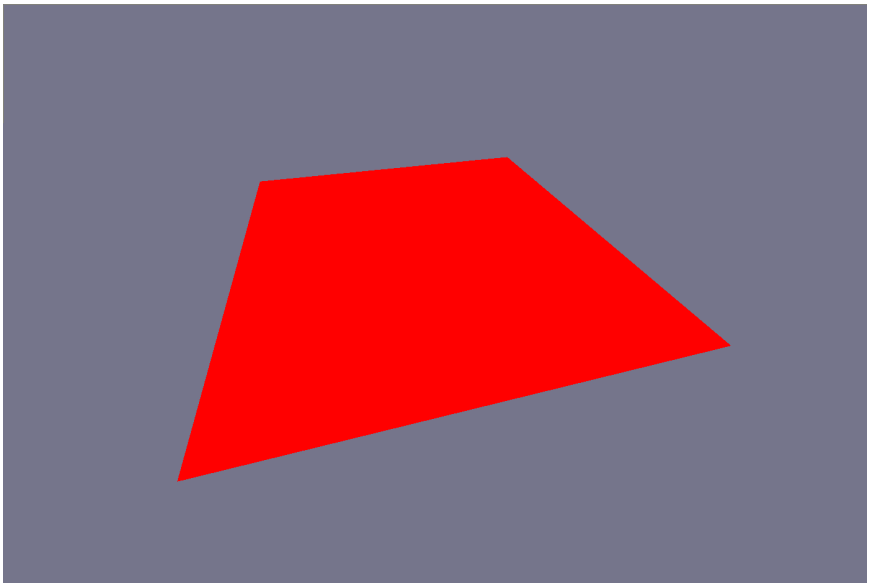


Figure C.1: **Smallflat**: A single plane of 20 by 20 patches.

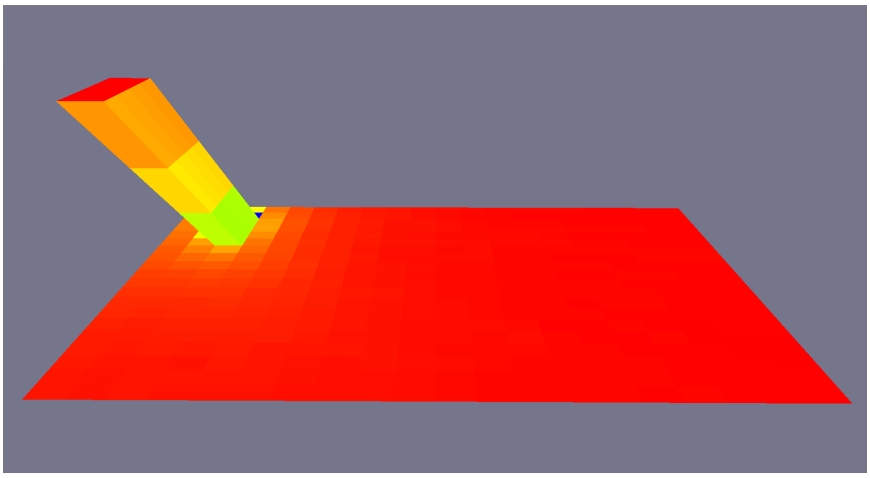


Figure C.2: **Nonuniformgrid**: A single building and ground with patch dimensions varying with direction. A patch is $1 \times 2 \times 3$ meters.

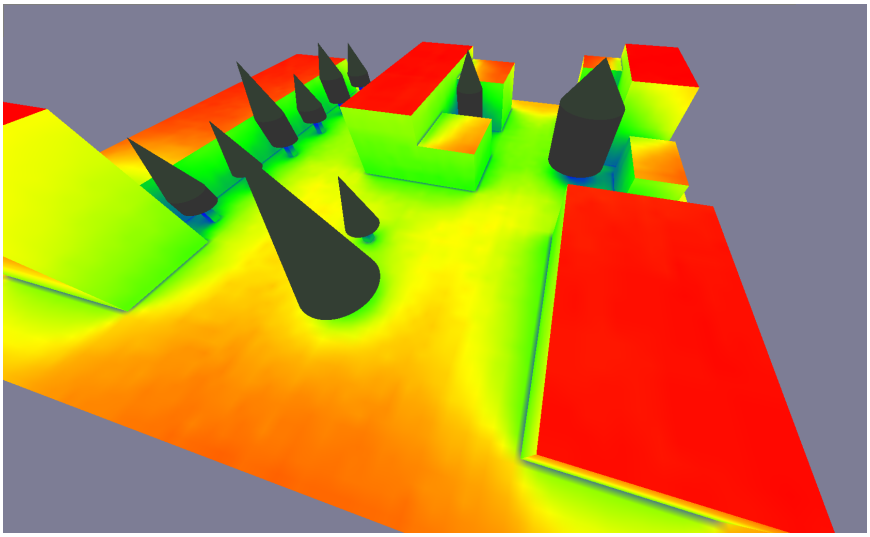


Figure C.3: **Urbanvege**: A sample urban domain with various trees. This test case is often used for illustrative purposes.

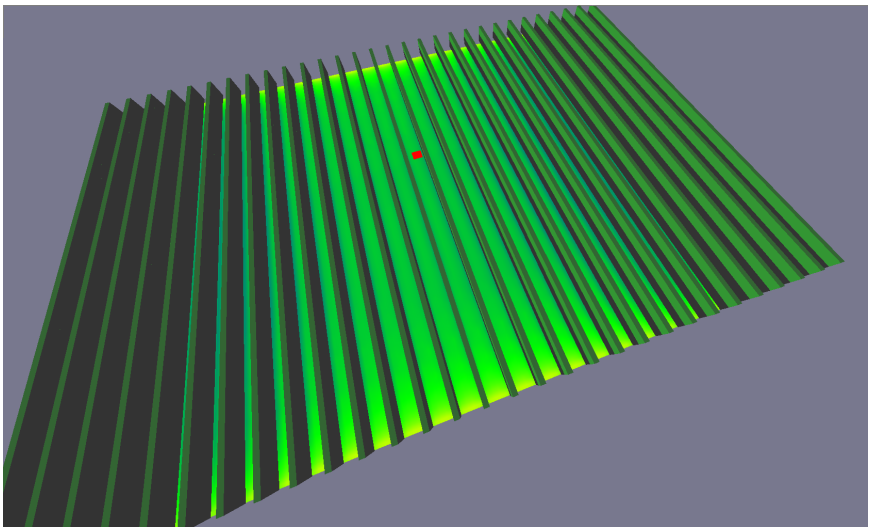


Figure C.4: **Vineyard**: A grape vineyard in Oregon. Contains a downward facing sensor to record reflected, scattered, and emitted radiation.

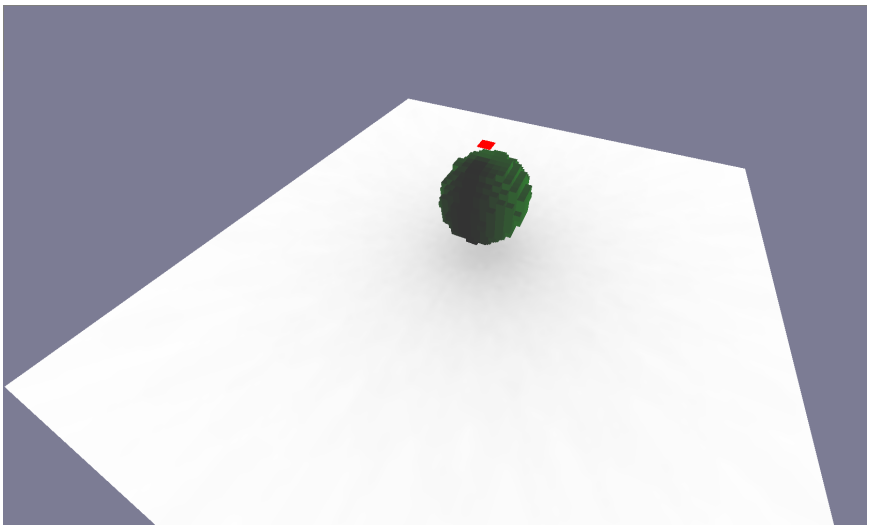


Figure C.5: **Isotree**: A high resolution isolated tree in a field of grass in Salt Lake City, Utah. Contains a downward facing sensor directly above tree crown to record reflected, scattered, and emitted radiation.

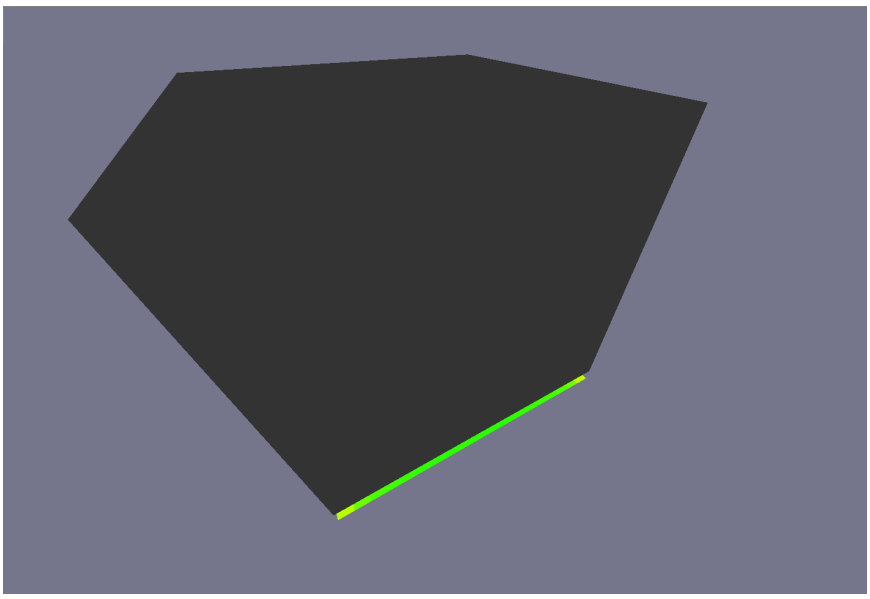


Figure C.6: **Densecanopy**: A $20 \times 20 \times 19$ block of vegetation volumes placed one meter above a ground surface. This scene can be used to "stress test" models.

D Hardware Details

D.1 Hardware

This section described the different machines and software settings used to compute results in chapter 4. These machines are referred to by their GPU and are defined as such.

D.1.1 Tesla S2050

OS:	Ubuntu 14.04.1 LTS
RAM:	516 GB RAM
CPU:	10x 8-core Intel Xeon E5-4650, 2.40GHz
GPU:	4x NVIDIA Tesla S2050, 3GB GDDR5 per GPU
Software:	NVIDIA OptiX 3.6.2, CUDA 6.0

D.1.2 GeForce GTX 690

OS:	Ubuntu 12.04 LTS
RAM:	16 GB RAM
CPU:	4-core Intel i7-3820, 3.60GHz
GPU:	NVIDIA GeForce GTX 690, 2GB GDDR5 per GPU
Software:	NVIDIA OptiX 3.0.1, CUDA 5.0