January 2015

Smart Cities: Inverse Design of 3D Urban Procedural Models with Traffic and Weather Simulation

Ignacio Garcia Dorado
Purdue University

Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Ignacio Garcia Dorado

Entitled

Smart Cities: Inverse Design of 3D Urban Procedural Models with Traffic and Weather Simulation

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Daniel G. Aliaga
Chair

Voicu Popescu

Dev Niyogi

Christoph M. Hoffmann

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Daniel G. Aliaga

Approved by: William J. Gorman 11/5/2015

Head of the Departmental Graduate Program Date

SMART CITIES: INVERSE DESIGN OF 3D URBAN PROCEDURAL MODELS WITH TRAFFIC AND WEATHER SIMULATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ignacio Garcia-Dorado

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2015

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I would like to thank all the support provided by my advisor, Prof. Aliaga, without him this PhD. would not have been possible. I am very grateful that he took me as his student from my first day at Purdue, for his restless help and guidance. I would like to thank Prof. Waddell for his encouragement to push on traffic simulation models, despite we could not finish, and the time I spent at U.C. Berkeley under his supervision. I would like to thank Prof. Benes with who, I wish I would have collaborated more extensively. Also, I would like to thank for the help and insight provided by Prof. Ukkusuri and Prof. Niyogi to develop state of the art simulators in passionate areas such as urban engineering and weather forecast. I have been very fortunate I came to Purdue and can develop such exciting work. I would like to thank Prof. Popescu, Prof. Hoffmann for being part of my dissertation committee, for helping to improve this dissertation and for the encouragement and ideas through the process. I would also thank my peers and friends in the CGVLab, starting with now Dr. Vanegas, he help me to discover what Computer Graphics was about in my early days at Purdue and raise me to the level of excellence he was use to. Also Gen Nishida, an excellent researcher and better person. I would also thank the rest of the members, including Ziang Ding, Jian Cui, Liang Li, Men-Lin Wu for his help the last years of my research. During my PhD., I have been funded by several agencies. I would like to especially highlight, Fulbright, they choose me as a scholar, that opened the door to Purdue and to meet incredible people, the Fulbright Purdue Association has been a essential pillar and a support all the way. I would also want to thank the National Science Foundation, the Purdue Research Foundation, U.C. Berkeley. I am grateful to Dr. David Luebke for his opportunity to work at Nvidia and the research we did there. I truly learned about computer graphics and grew my interest in rendering. I am very grateful for all the people I meet at Purdue. They help me

bear the demanding moments that a PhD. requires, to share the fun moments that the experience should have, and for listening and sharing with me this time that I will never forget. Specially thanks Aysegul, Alsu, Amina, Baiba, Eric, Gaurav, Gulcin, Heidi, Jon, Kate, Khalid, Ksenia, Lucie, May, Maricela, Mirisen, Yifei... and many more (the list does not fit here). Last but not least, I want to thank my family for everything they have done, their support (despite not being happy about me being far), their advice, and unbounded love has helped me to become the person I am today. For this, I thank with all my heart Jose Luis and Maria Jesus, the best parents anyone could even imagine, for the encouragement to study and to work hard, for opening the door to the imagination and the notion that everything can be achieved if you put your heart and soul on it. I also thank my siblings, Jose Luis and Beatriz, they are part of me even they are very far away.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

ABSTRACT

Garcia-Dorado, Ignacio Ph.D., Purdue University, December 2015. Smart Cities: Inverse Design of 3D Urban Procedural Models with Traffic and Weather Simulation. Major Professor: Daniel G. Aliaga.

Urbanization, the demographic transition from rural to urban, has changed how we envision and share the world. From just one-fourth of the population living in cities one hundred years ago, now more than half of the population does, and this ratio is expected to grow in the near future. Creating more sustainable, accessible, safe, and enjoyable cities has become an imperative.

A city cannot longer be seen as a static set of buildings interconnected with roads. It is both a complex and interdependent dynamic system. Many disciplines, such as urban planning, traffic engineering, and architecture, have created approaches that try to design and model different aspects of a city facing challenging problems. However, due to its massive scale and complexity there has not been any attempt to develop a framework that addresses all these aspects at the same time. This is our challenge.

We use an incremental approach to improve the realism of simulation and design: urban reconstruction, procedural generation, inverse procedural modeling, traffic engineering, and weather forecasting. We start with urban reconstruction that allows us to create detailed models for visualization and planning. Our methods focus on fast reconstruction and refinement of structures. Procedural modeling permits encapsulating the complex inter-dependencies within realistic urban spaces and enables users, who need not be aware of the internal details of the procedural model, to create quickly large complex 3D city models. Using machine-learning techniques, we can achieve real time interaction of high-level indicators. Vehicular traffic design has been carried out using aggregated simulations given that per-vehicle-simulation used

to be too computationally expensive. That is, after observing that each car's behavior depends just on its current state and surrounding vehicles, we discretize each lane in the road network as a set of contiguous memory bytes. At each simulation step each car concurrently can check its surroundings to define its future state. Finally, we explore how weather is an essential aspect of a city and how we can use it to improve its design. We develop a fast but complete weather simulator that allow exploring interactively how certain city designs exert positive impacts on a city when weather changes. We also present additional work in generating 3D assets from photographs for such urban modeling environments.

Initial applications of our work include creating enhanced and optimized 3D cities, simulating and visualizing urban space, traffic, and weather. Most of these applications have been developed in collaboration with academic, governmental, and industrial organizations. Our results include city models spanning up to 50 km^2, traffic simulation over 300,000 simultaneous cars, and weather encompassing 2500 km^2. We expect our efforts will ultimately increase the interdisciplinary collaboration in the development of better and smarter cities.

1. INTRODUCTION

1.1 Forward and Inverse Design: Procedural Modeling and Simulation

This dissertation presents a new computational approach to improving the realism, visualization, simulation, and design of a 3D urban area. A city is not a static set of buildings interconnected with roads; rather it is a complex and interdependent dynamic system. Many disciplines, such as urban planning, traffic engineering, architecture, and numerical weather prediction, have tried to design and model different aspects of a city. However, due to its massive scale, complexity, and interdependence between different parts there is not a current framework that combines these aspects.

In recent years, creating virtual environments has become an extremely important task for entertainment, urban planning, and training applications. This interest has sparked to seek new approaches to increase realism and new tools to quickly and easily design cities. In addition to the detailed modeling of complex urban geometry, some previous computer graphics work has also focused on the live aspect of the city. Works have provided methods to incorporate human behavior such as crowds simulation [1] and traffic simulation [2]. Other works have focused on increasing the realism through more accurate models and simulations of physical phenomena. For instance, CGI movies and games no longer rely on rough approximations of lighting (e.g., *Phong* lighting). Instead, they use physically based illumination with advances in ray tracing and global illumination [3]. Liquids are no longer single mock-ups, but rather complex realistic models [4, 5]. This demand has led to unprecedented levels of realism and interactivity, and the desire to quickly and easily capture existing urban spaces and to model new ones.

In this dissertation, we explore how urban procedural modeling can be enhanced with an inverse design of the geometry, traffic, and weather. Procedural modeling

permits encapsulating the complex inter-dependencies within realistic urban spaces and enables users, who need not be aware of the internal details of the procedural model, to quickly create large complex 3D city models. Using machine-learning techniques, we achieve real-time interaction with high-level indicators. We provide a tool for vehicular traffic design that uses a per-vehicle-simulation. Since each car behavior just depends on its current state and its surrounding vehicles, we discretize each lane in the road network into a set of contiguous memory bytes. At each simulation step, every car concurrently checks their surroundings and defines their future state and position. Moreover, we explore how weather is an essential aspect of a city and how we can use it to improve its design. We develop a fast but complete weather simulator for exploring how city designs and weather interrelate. Finally, we perform urban reconstruction and asset generation to create detailed models for visualization and planning. This method focuses on fast reconstruction and refinement of urban structures.

We hope our effort will ultimately help: i) urban planners to develop faster urban areas, easier to implement and higher control; ii) traffic researchers to create and optimize models using micro-simulators instead of macro-simulators and enhance its visualization; iii) weather researchers to explore more systematically and robustly the space of solutions; iv) all these fields to combine their efforts and create better cities. We also hope to benefit the development of virtual environments and digital models, not just within the research community but in entertainment, such as virtual worlds for video games and movies.

1.2 Procedural Modeling

Procedural modeling and more specifically, urban procedural modeling, has increased its popularity not just in computer graphics but other areas such as entertainment and planning applications. The main strength of procedural modeling is the detail amplification. Once the complexity of the model is encapsulated in a set

of *rules*, the system is capable of generating an infinite variability of complex and realistic models. Moreover, procedural modeling is starting to be used by the entertainment industry that demands endless realistic worlds for games and movies, making impractical that the artist can define the details manually. Finally, encoding the complex details into *parameters* increases the ability of potential non-expert users to quickly create large and complex models

In computer graphics, procedural modeling has focused on several domains such as plant generation [6], plant growth [7], terrain generation [8], and urban modeling. The seminal work by Paris and Müller [9] was the first in this class to use procedural modeling to create a forward procedural approach to generating detailed 3D models of cities. Starting with an L-system to generate the road network, blocks are extracted (i.e., the polygons formed by the road segments), the blocks are divided into parcels, and the parcels are filled with buildings. Later work has tried to better capture the complexity of such models (e.g., [10–12]). Effectively, the detail amplification inherently provided by procedural modeling is exploited: a small set of compact rules and parameters can yield very complex and coherent outputs.

However, the codification of the complex inter-dependencies in a small set of rules and parameters is also its Achilles' heel: i) controlling the generation of a specific model is a challenging task that requires in-depth knowledge of the procedural rule codification; ii) extending the system to generate different outputs, i.e., extend the set of rules, is very complex and usually requires expert users. Since there is not explicit control, the user must set the values for the input parameters, implement the procedural rules in software, and iterate between code and parameters to examine the output to achieve the desired model. Something that it can be used for few dozens input parameters but becomes impractical for more complex models defeating the advantage of detail amplification of procedural modeling. For the later, the user must explore in which domain the different output lies, one option is to add extra rules or modify the current ones, another option is to explore the input domain.

1.3 Inverse Procedural Modeling

Inverse procedural modeling addresses the aforementioned limitations by enhancing the model generation with inverse design tools. The design usually includes the control of user-specified values (*indicators*). These values include static variables (e.g., percentage of parks), average values (e.g., road utilization), and temporal variables (e.g., temporal behavior of clouds). The user selects the desired indicator values for the model or simulation. The inverse design engine finds the parameter values to generate such indicators values.

In recent years, there has been an increasing interest in this area. Šťava et al. [13] presented an innovative inverse system that focused on 2D content using an L-System. Bokeloh et al. [14] explored symmetries to complete models. Other works, such Aliaga et al. [15] and Vanegas et al. [16], focused on determining parameters for pre-specified classes of procedural building models. Several facade-level works have also proposed methods to determine procedural parameter values for individual facades (e.g., [17, 18]).

Our inverse engine attempts to control modeling by discovering how to alter the input parameter values and models to yield a desired set of user-defined target values while treating the urban procedural model as a black box for the purpose of generality. Many classical inverse methods are based on regularization theory (e.g., [19]); however they are designed for a different purpose than ours such as to remove noise and/or assume unknown formation process. Therefore, since the solution space is large, and the system is non-linear, our method is based on Markov Chain Monte Carlo (*MCMC*), more specifically on Metropolis-Hastings (*MH*) method. MCMC is a group of stochastic methods that sample probability distributions based on a Markov chain. We use MH to randomly walk the domain space using a probability distribution and different energy levels to find a solution. Depending on the design, our MC based method will have different variations: i) start from one single seed (to find a solution similar to the original) or from multi-seeds (to explore the solution space);

ii) optimize one variable (find the minimum) or optimize one variable but minimizing the cost of the change; iii) alter the model (to find a new combination of land use or procedural model), alter the initial conditions, or both. In our results, we show how these different variations can be used to find innovative solutions, designs, and optimizations.

My **thesis statement** is

> *An inverse computational method using a controlled randomized exploration of a large solution space for a nonlinear underlying system can be used to perform high-level and realistic design of the procedural generation of urban models incorporating complex geometry, vehicular traffic flow, and local weather simulation.*

1.4 Our Method and Summary of Results

Figure 1.1 presents the outline of our forward and inverse design. On one hand, in forward design the user defines a set of parameter values, the procedural engine generates a 3D model and a simulation is executed. The user can use that data for real-time rendering or, with our interactive tools, alter the parameter values to quickly see the model changes. On the other hand, in inverse design, the user defines a set of goal indicators. Our Metropolis-Hasting based method creates the models and runs the simulator to compute the indicator values and optimize the values until it finds the necessary parameter values to generate the desired model. Optionally, we use an artificial neural network engine (ANN) to replace the procedural engine and the indicator compute methods. This allows us to speed up the search process.

The set of desired model or behaviors is defined by a set of high-level *indicators*. The concept of indicators is well used by the urban design and planning community (e.g., [20,21]) but we extend it to encapsulate all outputs that our model or simulation can generate. Indicators provide an intuitive and often high-level means for the user to design and evaluate the model and its simulation. Depending on the context and

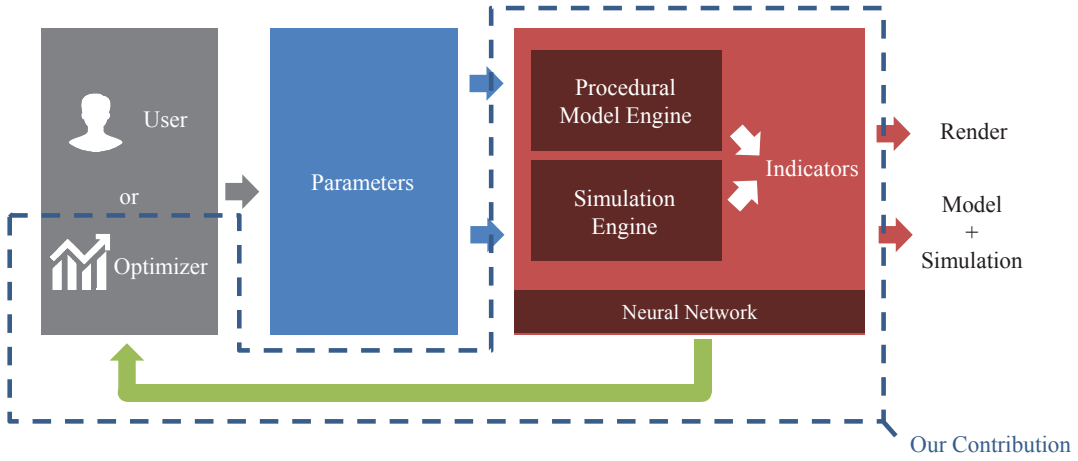

Fig. 1.1. **Our Inverse Design.** The user or an optimization tool controls the input parameters to create a city and optionally simulate to compute some output values or indicators. Using an optimization method, Metropolis-Hasting, we can find the desired behavior or model.

the purpose, the indicators can be as simple as a single value (e.g., temperature at the city center, distance to park) to complex outputs with semantic meaning (e.g., landmark visibility, rain intensity). Our MH method efficiently searches the high dimensional space to find such solutions. To our knowledge, indicators have not been used to control modeling or simulation because the exact relationship between the input parameters and target indicators is in general unknown, complex, and highly non-linear.

Advantages

Our forward and inverse design strategy has these main advantages:

- *Abstraction:* Procedural model encapsulates the complexity within rules and parameters. However, it is hard to have true control over the generated model or simulation. Our system adds a layer of abstraction, defining the goal variables as indicators that are easy to understand and control.

- *Interactivity:* Our approach enables interactively manipulating indicator targets while regenerating the model or simulating the scenario faster than real-time. This enables a new way to interact with the model and the simulators that were not possible before.

- *High-level and Detailed design:* High-level indicators can be used to control the model or simulator, this allows the users to quickly design complex models and simulations that have the desired appearance or behavior. Detailed design and low-level indicators define specific variables or behaviors. This allows the user to create customized designs or optimize models for highly control outputs.

- *Static and temporal design:* Indicators can be static variables that help design geometry or global variables. Temporal design allows the user define behavior over time to control it.

We explore three different areas to apply our method (Figure 1.2): 3D Urban Modeling, Traffic Simulation, and Weather Simulation. The ultimate goal is to design, control, and optimize an urban model, not just from the geometry perspective but from innovative areas such as traffic and weather. In this dissertation, we will modify the parameters and alter the models to present interesting results, this will illustrate how our technique can be used to allow the user have more control and create complex models quickly with the desired behavior.

Fig. 1.2. **3D Urban Modeling.** We show the 3D results of our three areas: 3D urban models, traffic simulation, and weather simulation.

3D Urban Models

We have created a forward and inverse design tool for 3D urban procedural models. This collaborative work with Carlos Vanegas, a former Purdue PhD student, we created solutions for forward and inverse design. On forward design, the user selects the value for a set of sixteen input parameters for each place-types (neighborhoods) of the model. These parameters define the procedural generation of roads, blocks, parcels, and buildings. On inverse design, the user controls this generation using high-level design indicators, for instance, landmark visibility or sun exposure, and geometry design indicators, for instance, distance to park or buildable area. Figure 1.3 shows an example we created of 3D urban design. In this case, the user defines an indicator that measures how dark/bright or compact/open a city is. This variable is computed as the amount of area in the city that is in shade. Using an ANN engine to speed up the process, we are able to use our MH based optimization method to find the necessary combination of input parameters to find the desired behavior at interactive rates.

Fig. 1.3. **3D Urban Design Example.** We show an example of design using a high-level indicator, in this case, open/bright vs. compact/dark city. Using a single slider the user can control it: a) high shadowing or compact, b) medium shadowing, and c) low shadowing or open.

Using our approach, the user just has to define the metrics that wants to control; then the system learns the necessary changes to control it. This idea is both useful for non-expert users that might not have the in-depth knowledge to achieve the desired

design, but also for expert users since our method allows to explore non-obvious and innovative solutions expanding the solution space.

Traffic Simulation

Designing and optimizing traffic behavior and animation is a challenging problem of interest to virtual environment content generation and urban planning and design. We create a super fast traffic micro-simulator to not just animate vehicles, but to design traffic pattern and optimize the 3D model. Figure 1.4 presents an example of this design. The user runs the simulation and wants to decrease the traffic of a specific area. Using our MH based optimization method, we can alter the 3D model, the distribution of jobs, or both, to generate a 3D model that has the user-defined behavior.

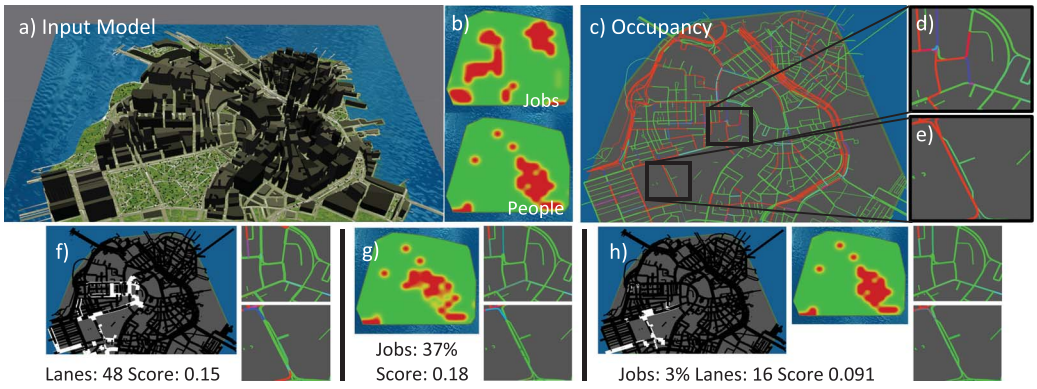

Fig. 1.4. **Traffic Design Example.** Given a fragment of central Boston a) and the distribution of job and people from a GIS source b), the user designs the traffic (occupancy) a desired area of the city c). Changing just lanes f), distribution of jobs g) or both h), our system learns the necessary changes to obtain the desired behavior.

We are able to encapsulate the complexity of a traffic simulation to be used in a procedural modeled city or a real world road network (from GIS data), design and control its behavior.

Weather Simulation

Weather is well recognized to be difficult to simulate and hard to predict due to its highly non-linear behavior, significant computational requirements, and need for precisely determined initial conditions (Pielke 2013 [22]). Hence, when designing content for a virtual environment, or for urban planning, the high input sensitivity, and computational cost make it very hard to simulate a realistic and desired general weather pattern. Thus, instead solutions in computer graphics typically script the visual appearance of weather phenomena in a manner similar to key-framing in animations creating unrealistic behavior patterns (for instance, rain is generated at the flip of a switch even if prior sky conditions are not indicative of rain); or, tools are used to automatically change the weather variables to control the shape and evolution of clouds without considering the actual behavior and state of the scene.

We create a physically based super fast weather simulation that has the necessary components to realistically model the behavior of weather. We use this weather model to create a set of tools to control and design the weather behavior. The user can define the 3D model and the initial conditions and create a simulation unbounded of time. We also provide tools to control and design the weather, with high-level control such as cloud coverage, to detailed control such as to control the rain and temperature of a city.

Figure 1.5 presents an example of our inverse weather design. The user interactively designs the model, selects the desired high-level behavior, and our optimization method finds the initial weather conditions necessary to generate such a weather pattern. Our system can also be used for the forward design, as well as to improve the design of an urban model.

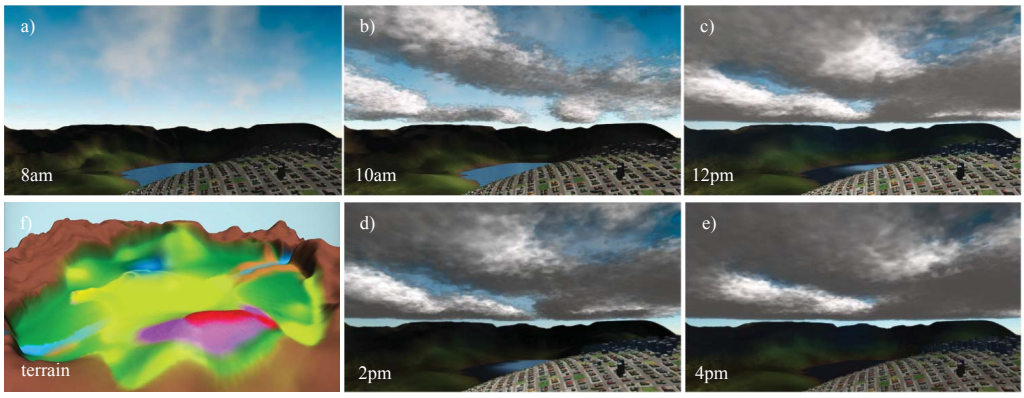

Fig. 1.5. **Weather Design.** f) The user draws the land use and designs the urban model. Then, the user defines a clear sky sunrise morning followed by afternoon showers (a-e) without providing details about realistic spatio-temporal behavior.

Reconstruction and Visualization

Finally, we present two methods to reconstruct the geometry of a 3D urban model. These 3D models are the response to the proliferation of urban planning, city navigation, and virtual reality visualization tools.

Figure 1.6 presents the result from one of our methods. Using GIS data, we are able to automatically obtain a volumetric reconstruction of the urban models. Using an innovative graph-cut algorithm, we are able to find the best texture to reconstruct a complex urban area. The generated models can be used for navigation (as Google Earth), urban planning, and visualization.

1.5 Main Contributions

The main contributions of this dissertation are:

- **Optimization Framework**: We present a computational framework to control and manipulate procedural and simulation models. We present a Metropolis-

Fig. 1.6. **Urban Reconstruction.** Starting from aerial imagery and GIS-style parcel/building data (left), we are able to automatically obtain a volumetric reconstruction (middle) to reconstruct a complex urban area (right).

Hasting MCMC-based method enhance with a two variable optimization and a time-invariant acceptance ratio.

- **Traffic:** We present a fast traffic micro-simulation engine including per-vehicle simulation, lane changing, car following, and intersection modeling. This allows to create a super fast visualization of traffic but also to create a traffic manipulation engine that enables specifying a desired traffic behavior and optimization.

- **Weather:** We present a super real-time weather simulation that includes temperature, wind, clouds, and rain. We use our method to visualize weather phenomena as well to provide an inverse design tool to control and design weather and optimize the 3D model of a city.

- **Interactive Design:** We create a new way to interact with the 3D model and the simulators. Our inverse procedural model hides the complexities of the rules, simulations, and parameters, to offer a clean and simple interface to draw, design, and control the procedural model and its simulation. Manipulating high-

level and detail indicators, the user can create complex and realistic virtual worlds within minutes.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 discusses the concept of procedural modeling and related work. Chapter 3 discusses our inverse design including our inverse procedural model, simulation methods (traffic and weather), and results. Chapter 4 discusses two approaches on how to reconstruct and visualize 3D urban models. Lastly, Chapter 6 provides conclusions and presents ideas for future work.

2. FORWARD DESIGN: PROCEDURAL MODELING AND SIMULATION

In this chapter, we describe the concept of *forward design*. In forward design (Figure 2.1), the user defines a set of input parameters (land use, procedural and simulation parameters), then the procedural engine and simulator creates a 3D urban model and outputs the simulation. There is not explicit control (since the user just has access to control the input parameters, the user manually tunes these parameters to generate the desired model and simulation. Note that for a non-expert user this might be a very complicated task.

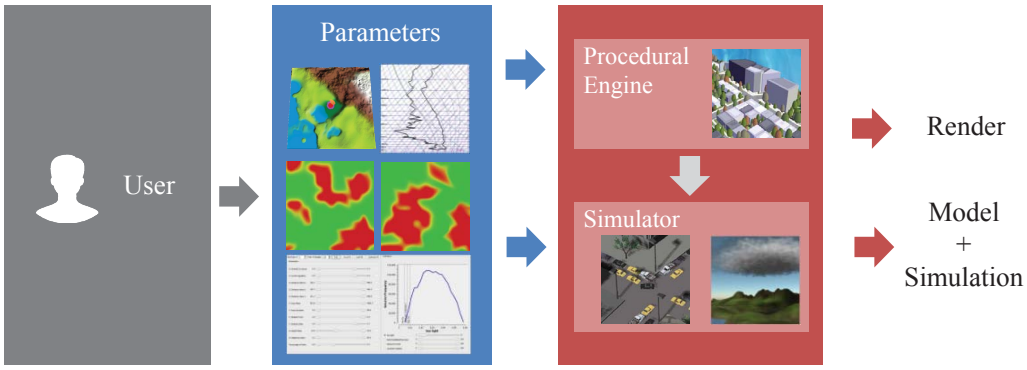

Fig. 2.1. **Forward Design.** The user defines the input parameters to generate the desired 3D model or create the desired simulation.

In the rest of the chapter, we present an overview of procedural modeling (Section 2.1), we describe our forward design tool for 3D urban procedural models (Section 2.2), and we describe our traffic (Section 2.5) and weather (Section 2.6) simulators.

2.1 Procedural Modeling

Procedural modeling is a technique that algorithmically creates a model or texture using a set of rules. There is a wide range of procedural modeling techniques, ranging from L-Systems to generative modeling. Depending on the purpose, the user can control the output with parameter values or adding or manipulating the rule set.

3D models can be created in three main manners: i) interactive modeling, ii) capturing, and iii) procedural modeling. In this dissertation, we focus on the second and third method, procedural modeling and capturing. Interactive modeling the user, usually an artist, use specialized software (e.g. CAD software) to model the 3D object. This interaction can be in the level of placing vertices and edges, or a higher level with geometry geometries. For realistic scenes, the models are very complex and contains millions of primitives. Modeling everything manually allows a complete control of the output but is extremely laborious and repetitive. Data-driven approaches use captured data to generate 3D models. These methods can range from aerial images to 3D laser scans of the objects. The main limitation is the dependence on scanning more objects to generate more variability for 3D environments.

2.1.1 Related Work

Procedural modeling focuses on generating 3D models algorithmically, typically using a small set of rules and data, that encapsulates the overall properties and complexities of the model. In computer graphics, it has been used to generate plants [6, 7], terrain [8], and urban modeling.

L-Systems

L-Systems was one of the precursors of procedural modeling and the first true procedural geometry model. Aristid Lindenmayer [6], a biologist, developed L-Systems to study the growth patterns of algae, but it has been extended to generate a mul-

titude of 3D objects. The idea is that every biological cell in a plant may divide simultaneously, so it is not enough to use a sequential model. L-System are defined with a grammar $G = \{V, S, \omega, P\}$ where V is the alphabet or set of symbols that can be replaced, S is the set of terminal symbols that are fixed, ω is the axiom, the string of symbols where the generation starts, and P is the set of production rules, they define how the variables can be replaced by other variables or by terminal symbols (note that if two rules can be applied, one will be chosen at random).

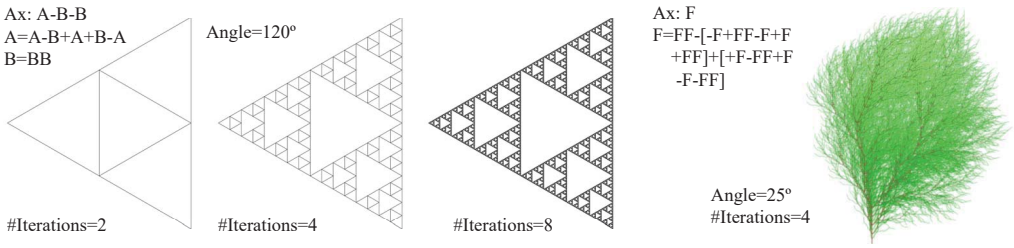

Fig. 2.2. **L-System.** (Left) An example of Sierpinski triangles for 2, 4 and 8 iterations. (Right) An example of plant grammar using two colors for stylization.

Figure 2.2 presents an illustration of two examples of L-System grammars. Using the axiom as starting point, the alphabet is replaced with the rules for a the *number of iterations*, this produces a string. This string is interpreted to be render. Usually, this is explained as the movements of a *turtle* that follow the path. The symbols meaning is as follow: $+$ the turtle turns left by an angle, $-$ turns right, F moves forward a d distance and $[,]$ means to go back one step. On the left, an example of an L-System to create Sierpinski triangles (a type of fractal). A complex structure can be created using two rules. On the right, another L-System example to generate a seaweed like structure.

Paris and Müller [9] created the first procedural approach to generating detailed 3D models of cities using L-Systems. Later works replaced L-system with *shape grammars* to generate the buildings.

Shape Grammars

Shape grammars are an extension of L-Systems where the production rules, instead of replace symbols, *shapes* are matched and replaced.

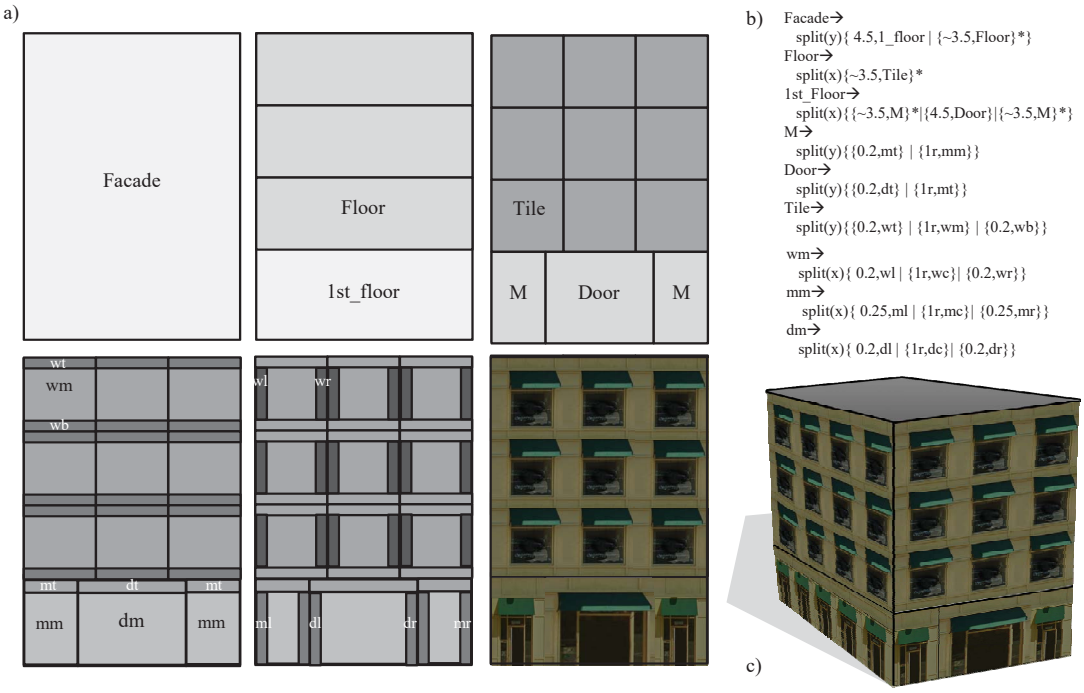

Fig. 2.3. **CGA Shape Grammar.** Production of a building with a CGA Shape Grammar: a) Facade production, b) CGA Grammar, c) final model.

For facades, Wonka et al. [23] presented a new type of parametric shape grammar, split grammars. These grammars restricted the type of rule to be applied and added control and attribute matching in 2D, this makes possible to generate a wide variety of building facades. The idea is to start from the axiom, the whole facade and apply production rules that take the current shape and replace it (split it) into several covering shapes. The process repeats hierarchically until all shapes are replaced by terminal symbols.

For buildings, Müller et al. [24] for CityEngine [25] extended the idea of split grammars to 3D with a new context-sensitive grammar, *CGA Shape grammar*. Figure 2.3 presents an example of such a grammar. A CGA grammar consists of an initial shape (in this case the facade) and a set of production rules. These rules are applied in order of priority from more general to more detail; this can be used to define different levels of LOD. A production rule is defined as $ID : PredecessorShape(cond) \rightarrow successor : (probability)$. For example, Figure 2.3b, the facade is split in the y axis creating one $1^{st} floor$ of 4.5m and as many floors as they fit $*$ of 3.5m ensuring that they are an integer value . This process is repeated, and the final model (Figure 2.3c) is created replacing the final rules with textures (or assets).

2.2 Our Procedural Engine and Simulation

We have implemented an urban procedural engine similar to previous city-level procedural modeling work (e.g., [9, 11, 12]). However, we provide a broad range of urban geometrical configurations with a reasonable degree of succinctness and high-level control. Our procedural engine is inspired by urban planners. For instance, our place-type categories and initial parameter values were obtained with the assistance of our urban planning collaborators.

Figure 2.4 presents an overview of the procedural generation. The user draws the terrain 2.4a and defines the input parameters. The procedural engine generates 2.4b the road network. From this network, blocks are extracted and divided into parcels 2.4c and vegetation is added. Finally, procedural buildings are created 2.4f.

Input Parameters

Our system counts with four sets of initial input parameter values (for the procedural generation and simulation): $\Omega = \{\omega_l, \omega_p, \omega_t, \omega_w\}$.

- The ω_l parameters (Section 2.3) refer to the percentage distribution of land use for each grid cell. For example, in the center of the city there is a nearly 100%

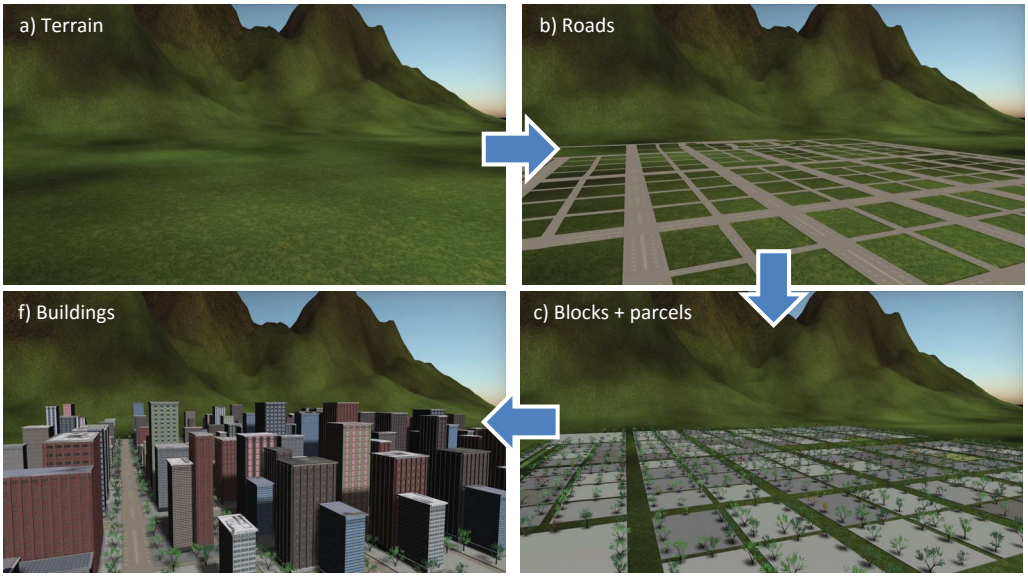

Fig. 2.4. **Procedural Modeling.** a) The user draws the land uses and define the terrain and b-d) uses our procedural engine to define the input parameters. Our procedural engine generates roads; c) blocks and parcels are extracted and random vegetation; d) the buildings are created.

likelihood of urban land use. This distribution can be defined with our interactive drawing tool, be loaded from a GIS data (e.g. WRF), or be procedurally generated.

- The ω_p parameters (Section 2.4) refer to the urban procedural parameters that define the urban geometry (e.g., building height mean, road width, percentage of white roofs).

- The ω_t parameters (Section 2.5) refer to the percentage distribution of people/jobs distribution of the urban area, as well to the road network enhanced with traffic parameters.

- The ω_w parameters (Section 2.6) refer to the initial conditions of the weather simulation. These conditions define the initial values for each grid cell for each

simulation weather variables. These conditions can be defined explicitly, procedurally, or via an observed *atmospheric sounding*.

Note all these parameters can be spatiotemporally varying. For instance, the atmospheric sounding can vary by either prescribing it from a weather model output and interpolation to each grid cell, or by using terrain and surface layer similarity/mixed layer similarity approximations [26] to re-estimate the sounding at each grid cell that is function of terrain, topography, and land use. In practice, we set the values initially, and we run the simulation.

2.3 Land Use

Using an interactive tool, the user 'paints' a distribution of land use categories over the terrain surface (ω_l), the terrain can be optionally altered. Relevant physical properties and initial conditions are given to our simulators. The spatial distribution of land use categories is used as input to our procedural modeling engine.

Our method supports the following twelve categories of urban and non-urban scenarios (based on typical usage in weather modeling):

- bare ground,
- desert,
- high mountains,
- grass (e.g., grass fields, prairies),
- forest (e.g., tree and dense/tall vegetation)
- snow, (e.g., typically on higher-elevation mountains)
- water (e.g., river, lake),
- crops (e.g., corn, wheat),

- low-density residential (e.g., houses),
- high-density residential (e.g., apartment buildings),
- low-density industrial (e.g., small commercial/industrial buildings), and
- high-density industrial (e.g., tall buildings, factories).

The properties of each land use depend on the application, but they contain a texture to display it and a set of variables that will be used for the simulations.

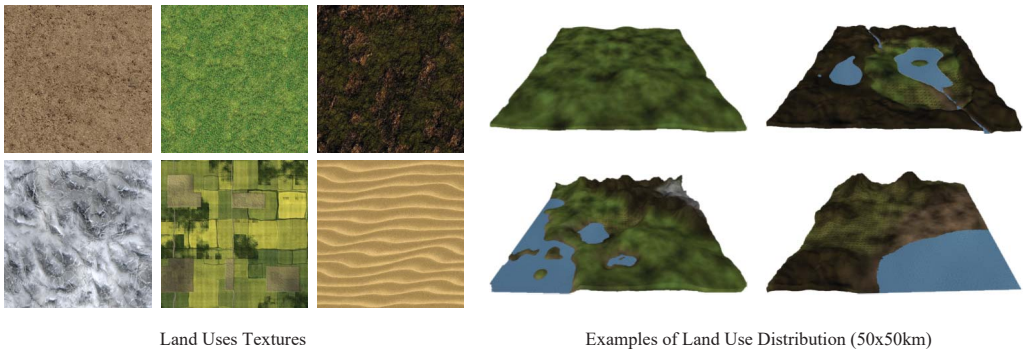

Fig. 2.5. **Land Use.** (Left) An example of land use textures: bare soil, grass, mountain, snow, crops, and beach sand. (Right) Several examples of land use distribution.

Figure 2.5 visually shows the idea of land use. On the left, several examples of land use texture. On the right, several examples of land use distribution. To generate these land uses distributions, the user can use our brush-like tool to draw the land use and/or elevation (this task never takes longer than one or two minutes) or load it from a file (we support several GIS data formats including WRF).

2.4 Urban Model

Urban procedural modeling is used to interactively define a set of *place-types*. Place-types is a key concept used in urban planning and modeling centers (e.g.,

[27–29]). The city model consists of several instances of one or more place-type categories. All instances of the same place-type category are regions – ranging from a few blocks to an entire neighborhood – that have contained roads, parcels, parks, and buildings with similar geometric attributes (e.g., road width, parcel area, building height). Similar to the urban layout editor of [30], place-types allow defining, moving, rotating, and resizing large subsets of the city at once and can be used to very quickly produce a sketch of the urban model. The underlying road network, subdivision into parcels, placement of parks, and definition of building envelopes per place-type instance is generated with a fully parameterized approach. Note that the place-type category defines a template that provides specific (initial) values for parameters but the user can interactively tune them. In our current implementation, our place-type categories include regional/town/suburban center, low/medium/high-density industrial, and residential, retail, park, and institutional areas, but the user can expand them manually tuning the parameters and saving the current parameter set as a new place-type category.

In weather, land use and place-types are combined in *local climate zones* (LCZ). LCZs describe the land use as well as the urban distribution using high-level descriptors.

The interactive session consists of the user sketching the land use and the global configuration of the urban area and then directly manipulate the m parameters.

1. a user defines the land uses with a brush-like tool and optionally draws the elevation (otherwise a random elevation map is creating following the land use distribution),

2. the specified land use creates a boundary shape for the urban area,

3. the user creates $z > 0$ place-type instances of one or more categories, each ranging in size from a few blocks to an entire city. , and

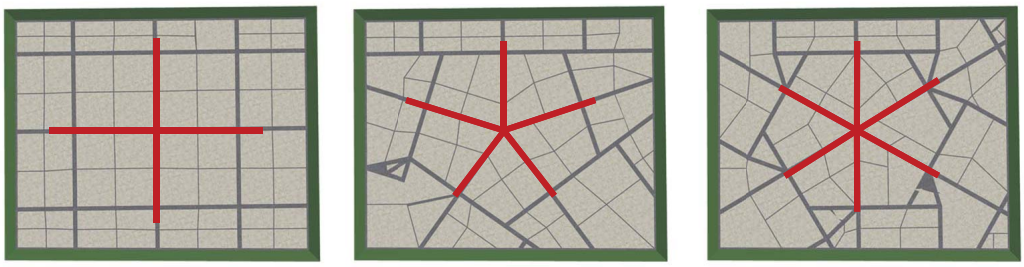

Fig. 2.6. **Number of Arms.** An example of road generation with different number of radial departing streets.

4. Once an instance of a place-type is positioned by the user, the geometry of the contained roads, parcels, and buildings is automatically created and joined with the neighboring geometry.

2.4.1 Urban Procedural Parameters

The entire 3D urban model has $m = zm_p$ parameters (ω_p) controlling its generation, with $m_p = 16$ being the number of per place-type instance parameters. The per-place type parameters generate a road network with two levels of street hierarchy (i.e., arterials and local), extract city blocks from the road network, subdivide the resulting blocks into parcels, define parks, and instantiate a 3D building envelope inside each parcel.

Place-Type. It is the higher order control of our system. It is defined by two rectangular radii: the main axis (u) and its orthogonal (v). The axes can be rotated using two handlers

- Center: It defines the origin of the place-type and becomes the origin of the road generation. It can be placed on any point inside the procedural area.

Fig. 2.7. **Parcel Area.** An example of parcel generation with different parcel area.

- Bounding Box: It is the place-type area of influence. It is defined by a bounding box with two main axis/directions (u and its orthogonal v) and two main radii. The main axis is initially aligned with North-South.

Roads. Roads grow radially-outward using the following road parameters (arterial and local roads have independent values):

- Block Size: Distance between two adjacent intersections.
- Length Irregularity: Random variability on the length of the roads segments.
- Angle Irregularity: Random rotation in an intersection respect the main axis.
- Number of Lanes: Number of two-directional lanes of the road.
- Number of Arms: Number of departing radial streets from the an intersection. Figure 2.6 illustrates the road generation for three different number of arms.

Blocks and Parcels. Blocks are extracted from the area enclosed by roads using a planar face traversal algorithm. Blocks are subdivided using recursive subdivision of oriented bounding boxes (OBB) to enforce that the parcels have access to the road. The OBB is computed for the current area to be subdivided (initially the complete block) and it is divided into two new parcels, the process repeats recursively until fit the following parameters:

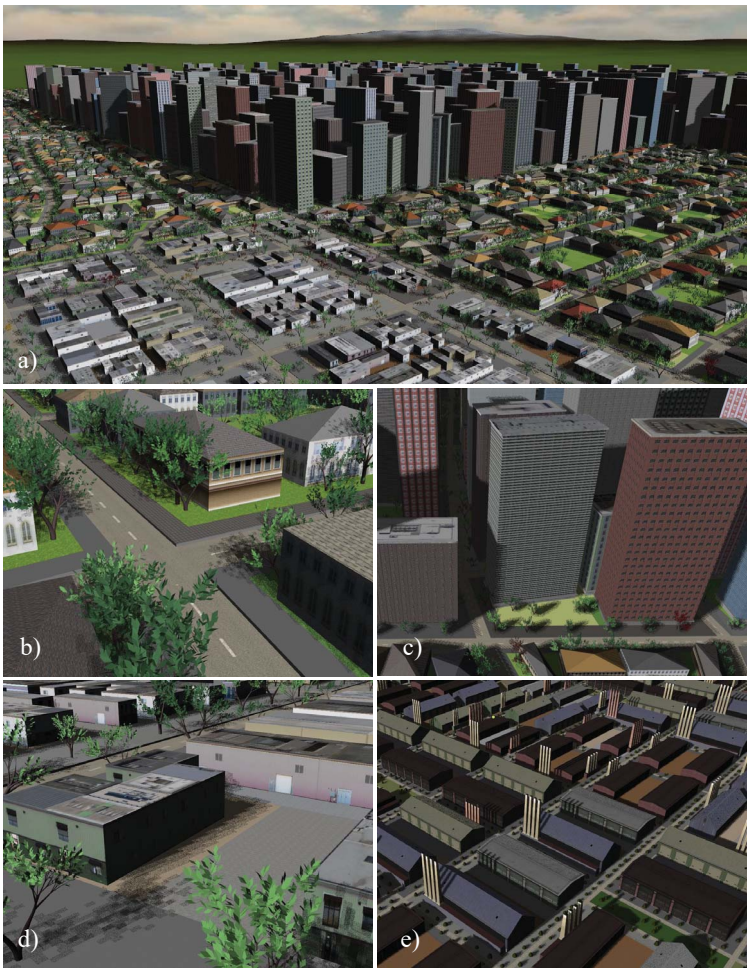

Fig. 2.8. **Urban Procedural Modeling.** Our method uses a) procedural modeling to generate a city and terrain, including b) low-density residential, c) high-density residential, d) low-density industrial, and e) high-density industrial.

- Parcel Area Mean and Deviation: Defines when to stop the recursive subdivision. Figure 2.7 illustrate the parcel generated for three different mean areas.
- Random Split: Offset of the subdivision split.
- Percentage of Parks: This can be done on block or parcel level.

Building. From the parcel, the building is generated using the following parameters:

- Building Type. Figure 2.8b-e presents the four different building types: Low-dense residential, high-dense residential, low-dense industrial, and high-dense industrial.
- Number of Stories and Deviation: Used in high-dense residential areas.
- Setbacks: Front setback defines the sidewalk width, side setback defines the distance between adjacent buildings.

2.4.2 Urban Forward Design

The user defines a set of these input parameter values $\{\omega_p\}$. $\omega_p = \{\omega_{p_1}, \omega_{p_2} \ldots \omega_{p_m}\}$ is the set of geometric input parameters to feed the procedural engine. Note that each parameter value must be within a range $[\omega_{p_{\min}}, \omega_{p_{\max}}]$. The user can alter interactively the input parameters to quickly generate a 3D urban model.

2.5 Traffic Simulator

Interactive modeling of urban spaces, with high realism and accurate behavior, is a fundamental challenge in computer graphics. Vehicular traffic is a ubiquitous dynamic activity in real-world cities which makes its simulation a necessity for realistic interactive urban environments. Moreover, with more than half of the world population living in cities, there is considerable interest in large-scale traffic simulation, design, and visualization. Virtual environment applications with a growing need for realistic vehicle traffic include virtual tourism, games and films, navigation services, traffic monitoring, eco-routing, and urban planning and re-design.

Traffic simulation and animation for computer graphics has received some recent attention (e.g., [2, 31–34]) but has also only been investigated in a forward fashion (i.e., simulate traffic for a given road network).

2.5.1 Related Work: Traffic Simulation and Roads

Procedural road modeling has received significant interest (e.g., [35–37]). However, traffic behavior is not simulated during design. Weber et al. [11] simulate traffic flow to estimate road widths and to improve a land use simulation. Their simulation only performs a stochastic sampling of a subset of all trips and estimates road occupancy. A detailed traffic microsimulation is not used. Traffic simulation/flow models can be categorized into the following three broad categories.

- *Microscopic models* simulate traffic interaction at an individual vehicle level including quantifying driver behavior, vehicle spacing, headway, speeds and lane changing (e.g., [38]SIM, MITSIM [39], [40]O, and [41]SUM). The key drawbacks are (i) agent-level calibration and (ii) large computational time.

- *Macroscopic models* provide aggregated representations of traffic (e.g., [42, 43]). Traffic is modeled as a continuum based on hydrodynamic kinematic wave equations using the fundamental relationships of speed, flow, and occupancy [44]. These models are faster but lack realism (and data) of individual vehicle behavior.

- *Mesoscopic models* simulate individual vehicles, but vehicle movement (or flow) is governed by macroscopic relationships rather than detailed per-car models.

People and Jobs Distribution

Our system requires storing the distribution of people and jobs over the urban area. While we could store people/jobs as spatially-located agents, it would be hard (and inefficient) to sample and relocate them. Instead, we store them as Gaussian probability distributions over a regular grid of cells – typically 200x200 meter cells. To sample the people or jobs distribution, we first randomly choose a cell proportional to its Gaussian probability distribution. Then, we find values within the cell by performing a 2D sampling of its Gaussian distribution.

2.5.2 Traffic Simulation

Traffic is simulated over many small time steps (e.g., $\Delta t \in [0.1, 0.5]$ secs). All our traffic-related models and parameters stem from well-known traffic simulation literature and are considered important in practice. First, our simulator executes per-vehicle trip planning. In each simulation step, a car's trip plan is used to update its position, velocity, and acceleration while inspecting the network, others cars, and intersections. Finally, we compute traffic performance metrics.

Trip Planning

If not provided as input, we augment the people/jobs distribution of the urban model with individualized trip plans consisting of a randomized schedule and desired route(s). Each person is assigned a home location and a job location. Starting at different times during the simulation period (e.g., 6 to 10 am), the vehicle departs its home location to reach the employment location and returns at a later time. To simulate trip chaining, for some vehicles, additional random destinations are added during the simulation period. To compute each vehicle's route, we approximately solve a time-dependent shortest path (TDSP) problem. Normally, solving the TDSP problem requires computing the effective travel time of each lane segment during all time steps within the simulated period. The effective travel time is used to update the shortest paths for all vehicles. This process repeats until the effective travel time does not significantly change (i.e., equilibrium). This methodology is very time-consuming and the update time is not bounded. In contrast, our approximate solution uses average travel time per lane (instead of a sampling of travel times during the simulated period), route-source grouping (subsection 2.5.2), and progressively-decaying route updating (subsection 2.5.2) which bounds the number of passes. The result is a fast approximate solution to TDSP (subsection 5.2).

Route-Source Grouping

The first simplification assigns each vehicle to the intersection closest to their source position (e.g., home, office). Dijkstra's shortest path algorithm computes the shortest path from a vertex to all other vertices. Thus, rather than executing Dijkstra for each vehicle during each update pass, we execute it only for each graph vertex (i.e., intersection) having at least one vehicle assigned to it. Hence, the number of executions of Dijkstra's algorithm is proportional to the number of vertices times the number of update passes. In practice, it is even less since only a fraction of the vertices corresponds to home or job locations. As an example, for a graph with 3000 intersections and 200,000 vehicles, we update all vehicles' shortest paths in only 0.09 milliseconds.

Progressively-Decaying Route Updating

The second simplification progressively reduces the number of vehicles that are updated during each pass of shortest path re-computation. After running the initial pass, although all vehicles are used in all passes we gradually reduce the percentage of the vehicles whose shortest paths are updated (e.g., 75%, 50%, 25%, 12%). This decay reflects that all vehicles do not necessarily follow the "best" shortest path and it also typically bounds the number of passes to 5 or less with reasonable trip planning.

Traffic Atlas

A given road network is converted into a compact 2D *traffic atlas* representing sampled road locations and an array of intersection records (Figure 2.9). One option to maintain the per-vehicle data for microsimulation is to maintain a set of lists. However, this requires sorting and queuing operations that are time-consuming for crowded roads. Since vehicles can be treated fairly independently, efficient memory access and easy separation of tasks is crucial for a parallelized implementation, our approach is to store vehicle data in a compact but parallel-access friendly traffic atlas.

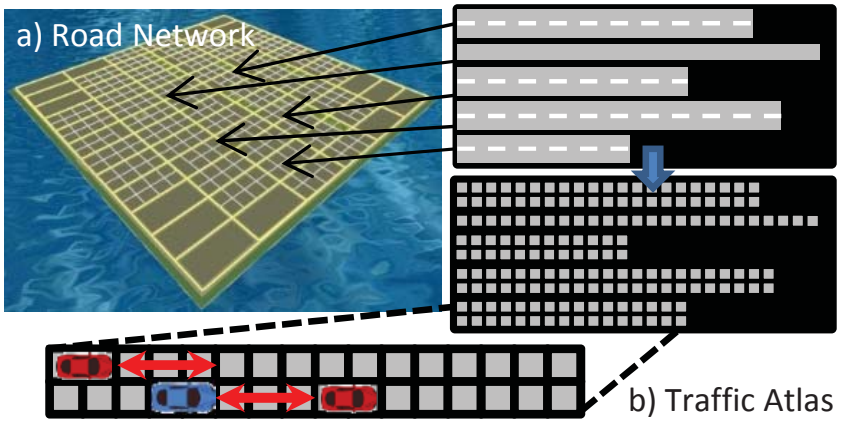

Fig. 2.9. **Traffic Atlas.** a) Each road lane is a row in the traffic atlas; b) the traffic atlas (top right) is sampled into delta segments (bottom right).

Our traffic atlas, akin to a texture atlas, compactly stores sampled road segments. Each road segment (i.e., graph edge) is stored as a set of rows (one per lane) of bytes where each byte represents t_m meters of a lane. Each byte can at most be occupied by one vehicle. The byte stores the car's speed (in m/s) times three (e.g., 55 mils/h or 88 km/h corresponds to 24.4 m/s and to the value 73). In practice, we found this quantization to be adequate. Intersections (i.e., graph nodes) are much sparser then sampled road segments and are stored separately. This data structure can efficiently store very large road networks (e.g., using $t_m = 1$ we can store 250,000 km of 4 lane roads in 1 GB of memory - this is roughly the length of all roads in Germany or Spain).

Simulation Steps

Given a set of per-vehicle trip plans and current traffic condition (traffic atlas), we update each vehicle's acceleration value and position, perform mandatory or discretionary lane changes, and update travel times.

Car-Following Model

Our simulation model is based on a discretized car-following principle: the current speed and acceleration depends on the distance to the following car (i.e., the next, or following, car in the direction of the current lane). As in Sewall et al. [2], we use the Intelligent Driver Model [45]. The acceleration/breaking function has two main terms:

- *free flow*: the vehicle's acceleration to reach its desired speed in absence of others (i.e., the speed limit), and
- *following-car closeness*: this term represents the deceleration when it comes too close to the one in front of it.

Altogether, we can write a vehicle's acceleration as

$$\dot{v} = a\left(1 - (v/v_o)^4 - (s^*(v, \Delta v)/s)^2\right) \tag{2.1}$$

where a is the acceleration ability of the car, v is the current speed of the car, v_0 is the desired speed, s is the distance gap to the following car, and s^* is

$$s^*(v, \Delta v) = s_0 + Tv + v\Delta v/2\sqrt{ab} \tag{2.2}$$

where s_0 is the minimum following distance, T is the desired time headway, and b is a comfortable braking deceleration. We follow the range guidelines from Treiber et al. [45] to assign random values to a, b, and T.

Lane-Changing Model

This model predicts when and where a vehicle will make a lane change based on two fundamental behaviors:

- *mandatory behavior*: the necessity of changing the lane in order to reach an exit or turn at an intersection.

- *discretionary behavior*: the desire to change lane in order to increase speed and bypass a vehicle.

Our model is based on a combination of the approaches of Yang and Koutsopoulos [39] and Choudhury et al. [46]. A vehicle always enters a new road in discretionary behavior and with an exponential probability might change to mandatory behavior. The probability to enter a mandatory behavior can be written as:

$$m_i = \begin{cases} e^{-(x_i - x_0)^2} & x_i > x_0 \\ 1 & x_i \leq x_0 \end{cases} \tag{2.3}$$

where m_i is the probability of vehicle i to be changed to mandatory behavior, x_i is the current distance from the vehicle to an exit/intersection, and x_0 is the distance of a critical location to the exit/intersection (e.g., last exit warning).

Gap-Acceptance Model

Once the vehicle has decided to change lanes, the maneuver is performed if the lead and lag gaps are acceptable. Using car speeds and distances, we compute the *critical lead gap* g_{ia}^D (i.e., minimum distance to the following car at which a lane change can be performed) and the *critical lag gap* g_{bi}^D (i.e., minimum distance to the lagging car at which a lane change can be performed):

$$g_{ia}^D = \max g_a^D, g_a^D + \alpha_{a1}^D v_i + \alpha_{a2}^D (v_i - v_a) + \epsilon_{ia} \tag{2.4}$$

$$g_{bi}^D = \max g_b^D, g_b^D + \alpha_{b1}^D v_b + \alpha_{b2}^D (v_b - v_i) + \epsilon_{bi} \tag{2.5}$$

where g_a^D is the desired lead gap for a lane change, g_b^D is the desired lag gap for a lane change, α is a system parameter (typically $\alpha = [0.05, 0.40]$) that controls the gap based on speed, v_i is the speed of the vehicle, v_a is the speed of the lead vehicle, v_b is the speed of the lag vehicle, and ϵ_{ia} and ϵ_{bi} are terms that add randomness to the behavior. We use typical values for these parameters obtained from [46]. If a vehicle's actual lead and desired gaps are within the critical and desired range and the lane changing mode is discretionary, a lane change occurs.

Simulation Update

During each simulation step, the traffic atlas, traffic lights, and vehicle information are updated. To avoid synchronization overhead and dependency on update execution order, we swap between two atlases by simply exchanging atlas pointers. Traffic lights are updated using round robin logic or using light phases and timings based on real-world data. The simulator checks if a vehicle is waiting to start a new trip. If there is room in the first segment of the vehicle's route, the vehicle is positioned on the traffic atlas with $v = 0$. The position and velocity of this now active vehicle, and all other active vehicles, is updated.

For each active vehicle, the simulator performs several update steps. First, it checks the distance to the following car. If none is found in the current lane, the simulator inspects the traffic signaling at the following intersection. If the traffic light is red, or it is the car's turn to stop at a stop sign, it corresponds to there being a following car at the intersection with $v = 0$. If the traffic light is green or it is not the car's turn to stop at the stop sign, our method finds the following car on the next lane segment of the vehicle's route. At a stop sign, the intersection tells the car when it can pass through. Second, given the distance to the following car (if any), the vehicle's acceleration, velocity, position and relevant traffic indicators (subsection 3.4.5) are updated. Third, the simulator considers vehicle lane changes (subsection 2.5.2). If a lane change is desired, the gap-acceptance model (subsection 2.5.2) is evaluated. Fourth, if an intersection is reached and it is not the destination, then the vehicle attempts to move to the next lane segment. If there is no space in the lane, it remains at the current location (i.e., $v = 0$).

2.6 Weather Simulator

In recent years, creating virtual environments has become an extremely important task for entertainment, education, urban planning, and training applications. This interest has fomented new approaches to increase realism and new tools to quickly

and easily design such environments. In addition to the detailed modeling of complex urban geometry, previous computer graphics work has also focused on the living aspect of the city. Some works have provided methods to incorporate human behavior such as crowds simulation [1] and traffic simulation [2]. Other works have focused on increasing the realism through more accurate modeling and simulation of physical phenomena (e.g., CGI movies and games use ray tracing and global illumination [3], complex realistic modeling of liquids [4, 5].

However, relatively little attention has been paid to the design and realism of weather phenomena in computer graphics. Previous works have provided methods for fog, rain, snow, and clouds. These methods focus only on rendering and not on the physical behavior of the phenomena. Moreover, todays graphics applications, including games, are no longer only short temporal sequences of minutes or hours, rather they can span much longer time horizons. Simulating weather is difficult because of its highly non-linear behavior, sensitivity to scale-dependent phenomena, and sensitivity to initial conditions [22]. Typical solutions in computer graphics script the visual appearance of weather phenomena similar to key-framing in animations (e.g., tools are used to control the shape and evolution of clouds [47–49] without considering the current atmospheric conditions). This can lead to unrealistic scenarios such as clouds and rain appearing at the flip of a switch even if prior sky conditions are not indicative of rain; clouds forming, at only one height in the atmosphere, even though there is no source of vertical motion; and a mishandling of the relationship between the land surface and the clouds (i.e. urban/rural heterogeneities); it also leads to error in the location of the clouds and that has a feedback error on other weather variables. These previous solutions are useful for small sequences where the goal is a complete control of the weather sacrificing realism.

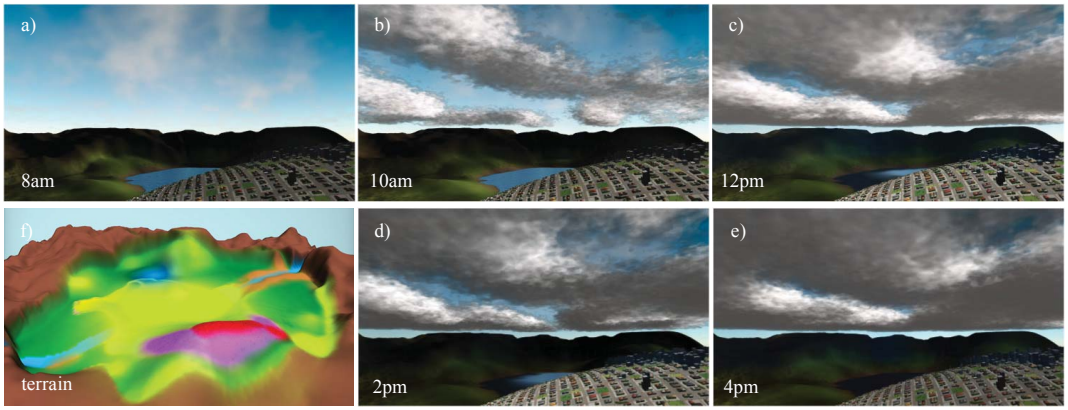

Fig. 2.10. **Urban Weather.** We present a method which tightly couples procedural modeling with a super real-time physically-based weather simulation. With our land use sketching interface (f), a user procedurally generates a terrain and city. Then, for example, our design tools enable intuitively choosing clear sky sunrise mornings followed by afternoon showers (a-e) without providing details about realistic spatio-temporal behavior.

2.6.1 Related Work

Cloud Simulation

The most closely-related simulations to our work in computer graphics are various forms of cloud simulation over time. Kajiya and Von Herzen [50] present an early method to solve the scattering equations and present equations which model the dynamic behavior of clouds. Dobashi et al. [51] present computationally inexpensive cellular automata to model cloud evolution using several simple transition rules and offline rendered clouds. Miyazaki et al. [52] use a coupled map lattice to approximate the formation of various cloud shapes. Overby et al. [53] modify a fluid solver to simulate clouds based on buoyancy, relative humidity, and condensation. Harris et al. [54] describe a diagnostic cloud simulation engine. Dobashi et al. [47] enable a user to draw the contour of a cumuliform cloud from a specific camera position and their system automatically adjusts parameters so that the simulated result fits in

the drawn contour. Harris et al. [54] does not have a prognostic simulation of cloud variables, radiative transfer model, rain variables, nor procedurally generated models of land use. Dobashi et al. [47] assume constant heat from the ground and do not take into account wind, radiation, and surface energy balance.

In contrast to and building upon the above methods, our approach is based on a full weather simulation model that includes surface energy balance, boundary layer processes, and first order evolution of meteorological forcing equations [55]. Also, most prior approaches are diagnostic. i.e. weather is static input that does not change, and the objective is to get the visualization of cloud and other phenomena are not simulated or interactively responding. This interactivity becomes important for scenarios where a priori specification or clouds or environment is not an option. Examples include where the graphics generate as part of a decision system in a serious game environment, battlefield theater environment or for education modules. In each of these examples, the decision either by the model or the user is intimately tight to the interactive graphics that emerges as a result of the environmental and initial conditions providing such model to the graphics community is one of the goals of this paper. Moreover, we provide very fast simulation performance.

Weather Forecast Models

Clouds and precipitation events are one of the most difficult features to accurately simulate in a coupled prognostic model. This is because their simulation and prediction require all other factors to be adequately represented and simulated, and errors in any of the interacting variables can translate into inaccurately developing clouds or numeral instability. As a result for most NWP models, rainfall and cloud forecast verification is considered as integrated performance of the model [56].

There are a very long history and a large body of work attempting to model and predict weather phenomena. Nebeker [57] and Stull [58] provide a comprehensive review and history of Numerical Weather Prediction (NWP). Weather forecasting can

be done at a variety of scales ranging from global-scale simulation (e.g., jet streams) to micro-scale simulation (e.g., wind effects in urban canyons). For our goal, we seek a physically-based, interactive local urban-scale simulation that models weather over a city and a range of land uses.

Implementations of several well-known weather forecast models are available. The Weather Research and Forecasting (WRF) Model [59, 60] is now the state-of-the-art NWP system designed to serve both atmospheric research and local operational forecasting needs. The Regional Atmospheric Modeling System (RAMS) [61] is a mesoscale simulator (e.g., for horizontal scales of up to $2000km$ and grid cell sizes of up to $20km$) for weather and climate research and NWP. Though in our tests WRF is faster than RAMS, both systems are extremely computationally expensive. For example, a 72-hour WRF simulation for a 50x50km area at 1x1km grid cell size takes around ten hours using a computer server of 48 cores.

2.6.2 Weather Model

According to Stull [58] and Durran [62], a weather model at urban scale must account for multiscale feedbacks that span turbulent energy exchange, to micro and mesoscale energetics and dynamics. Thus, the model needs to consider modules to simulate features such as advection, diffusion, buoyancy, moisture processes (e.g., evaporation, condensation, auto-conversion, accretion) and radiation and surface energy balance processes. The physical properties of the land use categories, such as albedo, the Bowen ratio, and roughness, dictate the nature of radiative and turbulent surface fluxes that define the land-atmosphere convection interactions. Changes in the surface energy balance due to land surface heterogeneity produce micro/mesoscale circulations and a zone of local convergence. This heterogeneity induces convergence/divergence, this can create non-classical circulations akin to a land-sea breeze that produces vertical motion to lift the air directly or create local zones of convergence and divergence, which act to drive the air upward at the surface. To simulate

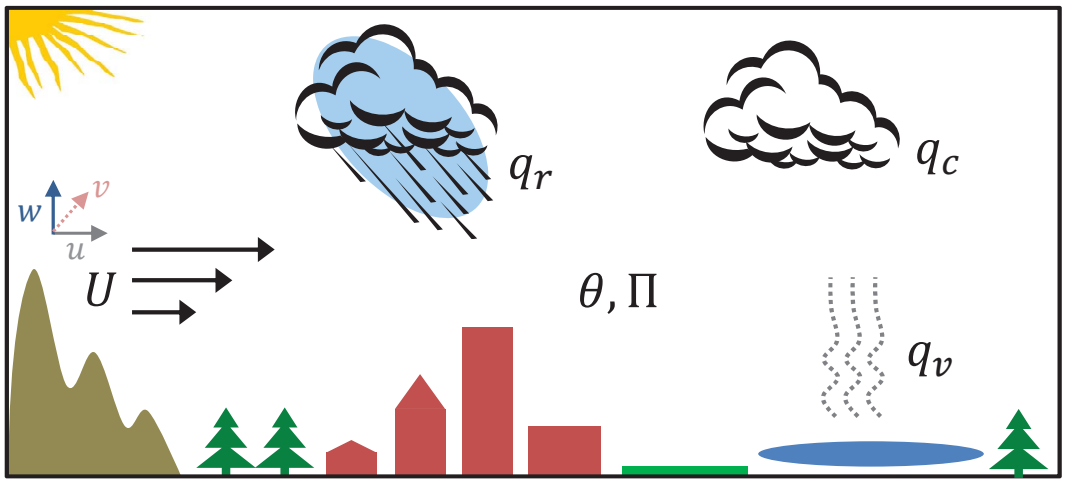

Fig. 2.11. **Weather Variables.** We show a depiction of the weather variables computed by our simulator.

these processes, the weather model makes use of the aforementioned physical properties and a set of weather variables stored in grid cells. This information is then passed on to a nonlinear system of dynamical equations evaluated over space and time.

Variables and Grids

Our approach uses the following variables stored within a 3D grid structure over the simulated region (Figure 2.11):

- **Wind velocity** ($U = \{u, v, w\}$): These are the wind components (measured in m/s) in the west-east, north-south, and vertical direction, respectively.
- **Potential temperature** (θ): The potential temperature of a grid cell of air is the final temperature the cell would have attained if it were moved adiabatically (i.e., without heat transfer) from pressure p and temperature T (in Kelvin) to

a standard pressure $p_0 = 100kPa$ (i.e., approximately sea level pressure). Its definition can be written as

$$\theta = \frac{T}{\Pi} \tag{2.6}$$

- **Exner function** (Π): The Exner function is a non-dimensional pressure quantity designed to simplify pressure-related computations in atmospheric modeling. It is written as

$$\Pi = \left(\frac{p}{p_0}\right)^{R_d/c_{pd}} \tag{2.7}$$

where R_d is the specific gas constant of dry air ($R_d = 287.058J/(kg \cdot K)$) and c_{pd} is the heat capacity of dry air at constant pressure ($c_{pd} = 1004.5J/(kg \cdot K)$).

- **Moisture variables** (q_v,q_c, q_r): These are mixing ratios of water vapor (non-dimensional quantities of mass-of-air per mass-of-air).

Our method uses the Arakawa C-Grid [63], commonly used in the weather community, to offset mass and energy variables in vertical and horizontal directions. A C-Grid is preferred for weather models because it prevents physically unrealistic waves of non-wave variables (i.e., temperature) from forming in the model, and performs better for second order differentiation [64]. In such a grid, the scalar variables are defined in the center of each grid cell, and the vector variables are prescribed in the faces. In the horizontal plane, the ground is split into evenly spaced grid cells typically of 1x1 kilometer. In the vertical direction, the grid spacing is log-linear with higher resolution closer to the surface within the boundary layer and then gradually. Thus, the user defines the first grid cell height d_z and the rest is calculated following the power law

$$z[k] = min(d_z \cdot s_r^k, d_{max}) \tag{2.8}$$

where d_z is typically $50m$, stretching ratio s_r is usually 1.025, and the maximum height d_{max} is $1000m$. This approximation was based on several realizations of the Monin-Obukhov similarity theory based profile estimation. For each horizontal grid,

this creates 12 grid cells under $3km$ (i.e., in the region that is expected to be influenced by the boundary layer) and the vertical extent of the topmost grid cell is about $25km$.

The model can be initialized using a realistic sounding at each grid location from an external source from reanalyses [65] or through a single sounding that can be interpolated to each grid location based on surface characteristics. These soundings are representations of the model variables converted from an observed atmospheric profile of temperature, moisture, and wind and are specified by our design tool. The lateral boundaries of the grid are assumed periodic (i.e., toroidal in 3D). The bottom boundary conditions are determined from the land surface model, and the top boundary conditions are described in Section 2.6.3.

Dynamical Equations

Our nonlinear dynamical equations consist of three components (expanded upon in Section 2.6.3):

- *fundamental modeling:* accounts for advection, diffusion, and buoyancy of wind, potential temperature, and Exner function.
- *cloud and precipitation modeling:* accounts for evaporation, condensation, autoconversion, accretion of water, and potential temperature.
- *radiation modeling:* accounts for short-wave and long-wave radiation onto the ground and urban surfaces, and the resulting potential temperature changes.

These three components are the minimum set needed to create a physically-based simulation that models temperature, wind, clouds, and rain. Note that without the radiation model, the system would reach an equilibrium after which there would not be weather changes during the day and night, and there would not be buoyancy (e.g., water vapor movement) and consequently no clouds. Without the cloud and precipitation model, there would not be clouds nor rain. Further, although the temperature, pressure, and wind might change, the effect on weather would not be very noticeable.

Finally, the equations of the fundamental model are crucial to tie the components together. Thus, each of the model components feeds into and build off the energetics of the different model components and are completely interactive. Indeed, this is the nature of the weather processes seen in nature as well.

2.6.3 Urban Weather Simulation

Given a set of initial values either calculated or provided, our weather simulator updates grid cell variables during each time step integrating finite difference partial differential equations described in this section.

Fundamental Component

Our set of fundamental equations are derived from the laws of motion and thermodynamics and refined using scale-analysis for our desired urban-scale weather simulation. The equations are as simple as possible to reduce computational expense yet still yield relevant weather phenomena. The fundamental equations model the motion of the wind (U), temperature via potential temperature (θ), and atmospheric pressure and density via the Exner function (Π). The equations simulate advection, diffusion, mass and energy balance, and buoyancy [55]. Advection (e.g., $U \cdot \nabla$) is the transport mechanism of a substance by a fluid due to the mean flow motion. For instance, liquid water that forms the cloud droplets is diffusion/turbulent exchange and advection together cause the vertical and lateral exchanges. Diffusion (e.g., $\kappa\nabla^2$) represents molecular and turbulent exchange of a substance from a region of high concentration to a region of low concentration. For a parcel of air in hydrostatic balance, there is a balance between gravity and the pressure gradient force and hence air does not move vertically. However, if there is a change in density (e.g., in ideal gasses due to a change in temperature or pressure) a buoyant force is exerted. If the parcel is less dense (than the surroundings) the parcel exerts an upward force; if it is denser, then the net force is downwards. As a first test, we will show that a cold

bubble sinks and a warm parcel of air rises (until it reaches a state of equilibrium), and have hydrostatic and dynamical equilibrium built-in.

For brevity, the material derivative is defined as the sum of the local change in a variable with the advected portion of that variable: $\frac{D}{\partial t} \equiv \frac{\partial}{\partial t} + (U \cdot \nabla)$. To improve numerical stability, each variable is decomposed into its base value and a time-varying perturbation value (e.g., $\phi = \phi_0 + \phi'$ where ϕ is the instantaneous value of a vector or scalar variable, ϕ_0 is the base value, and ϕ' is the perturbation as a result of external forces). This allows us to simplify some formulas to save computational cost. To illustrate the conversion of the equations to actual code, we have added an example of a complete equation with its discretization is provided in Supplemental Material A.

Wind. The equations modeling the change of the horizontal wind velocities u and v are

$$\frac{Du}{\partial t} = -c_{pd}\theta_\rho \frac{\partial \Pi'}{\partial x} + \kappa \nabla^2 u^{t-1}, \tag{2.9}$$

$$\frac{Dv}{\partial t} = -c_{pd}\theta_\rho \frac{\partial \Pi'}{\partial y} + \kappa \nabla^2 v^{t-1}, \tag{2.10}$$

The right-hand side of Equations (2.9) and (2.10) include two terms:

- The first term represents acceleration due to the pressure gradient force expressed by the Exner function and θ_ρ is the density potential temperature which accounts for both liquid and vapor water in air density.
- The second term represents wind diffusion based on the winds values in the previous time step and by a constant factor κ.

The equation modeling the change of the w wind velocity (i.e., vertical wind) at time t is

$$\frac{Dw}{\partial t} = -c_{pd}\theta_\rho \frac{\partial \Pi'}{\partial z} + g\left(\frac{\theta'}{\theta_0} + 0.61 \cdot q'_v - q'_c - q'_r\right) + \kappa \nabla^2 w^{t-1}, \tag{2.11}$$

where the right-hand side includes a first and third term similar to the u and v wind velocities. The second term is the computation of vertical buoyancy as a potential

temperature of a parcel (perturbation) compared to its surroundings (mean state) with moisture added to represent acceleration from falling liquid water.

Pressure. The Exner function value is updated using the following prognostic equation

$$\frac{D\Pi'}{\partial t} = \frac{c_s^2}{c_{pd}\rho\theta_\rho^2}\nabla\cdot(\rho\theta_\rho U) + \kappa\nabla^2\Pi'^{t-1}, \tag{2.12}$$

where $c_s = 100m/s$ is the artificially lowered inelastic speed of sound and ρ is air density (in kg/m^3). The first term computes advection limited by the speed of sound. The second term computes pressure diffusion. Note that while mathematically valid but physically unrealistic, a solution to an atmospheric wave equation is sound waves. However, since sound waves do not propagate energy on the same scale as wind advection, they are artificially eliminated to preserve model compressibility and also reduce computational cost.

Temperature. The potential temperature perturbation values are updated using

$$\frac{D\theta'}{\partial t} = -\rho w\frac{\partial\theta'}{\partial z} + \kappa\nabla^2\theta'^{t-1}. \tag{2.13}$$

The first term represents the change in temperature due to non-adiabatic vertical advection and the second term represents its diffusion.

Clouds and Precipitation Component

To simulate clouds and precipitation at urban scale, we build off the conceptual approach use previously in Pielke et al. 2007. In our current implementation, we focus on convection and warm rain process. We use the classic equations derived from Kesslers microphysics scheme [66] and the mesoscale implementation framework of Soong and Ogura [67].

Microphysics Scheme

Kesslers is a warm (liquid-only) cloud scheme that includes water vapor, cloud water, and rain. The equations use a variety of constants determined experimentally.

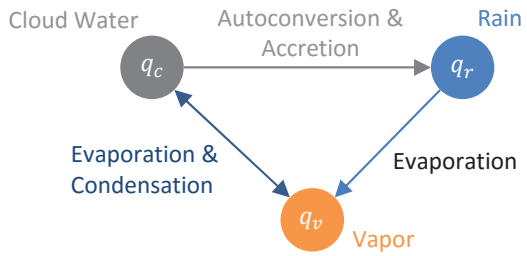

Fig. 2.12. **Kessler Microphysics.** State transitions of water in the clouds

Kesslers scheme is a popular option for representing cloud physics. In this schema cloud formation and dissipation are estimated through condensation and evaporation, and rain drop growth via the growth of cloud droplets (autoconversion) and the collection of droplets from falling rain (accretion). Figure 2.12 depicts the model framework.

Condensation. This is the process by which water vapor in the air is changed into liquid water and handles cloud formation and precipitation. The expression for condensation from vapor (q_v) to liquid (q_c) is implemented based on Tetens [68] as

$$C_{v\mapsto c} = \min\left\{q_v, \frac{q_v - q_{vs}}{1.0 + q_{vs}\frac{a(273-36)L_{lv}/c_{pd}}{(\Pi\theta)^2}}\right\} \tag{2.14}$$

where $a = 7.5ln(10)$ and the constant term $L_{lv} = 2.501 \cdot 10^6 Jkg^{-1}$ corresponds to the latent heat of vaporization. The use of the minimum operator ensures mass balance between water vapor (q_v) and liquid water (q_c). The equation also makes use of the theoretical maximum amount of water vapor that air at a specific temperature and pressure can hold. This is known as the saturation mixing ratio and is implemented in our model as

$$q_{vs} = b\exp\left(a\frac{\Pi\theta - 273}{\Pi\theta - 36}\right) \tag{2.15}$$

where $b = 380.16/p_e$, and $p_e = p_0\Pi^{c_{pd}/R_d}$.

Evaporation. This can be considered as an inverse process of condensation: liquid water (q_c and q_r) turns into water vapor (q_v). The model considers the air saturation potential and when the air becomes unsaturated, cloud droplets (q_c) evaporate to maintain a balance between liquid water in the atmosphere and water vapor. Thus, the rate of evaporation is self-regulated so as to keep the air at the saturation mixing ratio until the cloud droplets are totally evaporated. Raindrops (q_r) evaporate after cloud droplets are condensed by the moisture deficit. Evaporation is calculated as

$$E_{c,r\mapsto v} = \begin{cases} q_r \\ -C_{v\mapsto c} - q_c \\ V_c \frac{q_{vs}-q_v}{\rho q_{vs}} \frac{(\rho q_r)^{0.525}}{5.4\cdot 10^5 + 2.55\cdot 10^8/(\bar{p} q_{vs})} \end{cases} \tag{2.16}$$

The top term constraints evaporation does not exceed the rain, the middle term ensures the newly available vapor is not exceeded, and the third empirically determined term ensures the rate maintains the saturation mixing ratio. The third term also uses the ventilation coefficient V_c which is used in determining air pollution concentration near the surface ground. It is calculated by the experimentally-determined equation

$$V_c = 1.6 + 124.9\rho q_c^{0.2046}. \tag{2.17}$$

Autoconversion. This process computes rain droplets to be formed if cloud water exceeds a critical value. Its equation is

$$A_{c\mapsto r} = \begin{cases} 0.0 & \text{if } q_v \leq q_{c0} \\ \max\left(0.0, k_1(q_c' - q_{c0})\right) & \text{if } q_v > q_{c0} \end{cases} \tag{2.18}$$

where conversion factors $k_1 = 10^{-3}s^{-1}$ and $q_{c0} = 10^{-3}gkg^{-1}$.

Accretion. This last process represents when rain collects the small cloud droplets while falling. It can be approximated by

$$B_{c\mapsto r} = \max\left(0.0, k_2 \cdot q_c' \cdot q_r'^{0.875}\right), \tag{2.19}$$

where $k_2 = 2.2$ according to Kessler [66].

Model Implementation

Finally, we put together the previously described different microphysics concepts to write a set of per-grid cell water equations, representing clouds and precipitation, which conserve total net mass as well:

$$\frac{Dq_v}{\partial t} = -\rho w \frac{\partial q_v}{\partial z} + \kappa \nabla^2 q_v - C_{v \mapsto c} + E_{c,r \mapsto v}, \tag{2.20}$$

$$\frac{Dq_c}{\partial t} = \kappa \nabla^2 q_c + C_{v \mapsto c} - A_{c \mapsto r} - B_{c \mapsto r}, \tag{2.21}$$

$$\frac{Dq_r}{\partial t} = \kappa \nabla^2 q_r + \frac{1}{\rho} \frac{\partial \rho V_t q_r}{\partial z} + A_{c \mapsto r} + B_{c \mapsto r} - E_{c,r \mapsto v}, \tag{2.22}$$

where $V_t = 36.34 \sqrt{\frac{\rho_0}{\rho}} (\rho q_r)^{0.1364}$ is the rain water terminal velocity and ρ_0 is the density of the lowest grid cell per column. These equations also make use of the material derivative (as in Section 2.6.3). Considering the energy change associated with water phase change (evaporative cooling, condensation heating), the thermodynamic equation is updated (instead of Equation (2.13)) as

$$\frac{D\theta'}{\partial t} = -\rho w \frac{\partial \theta'}{\partial z} + \kappa \nabla^2 \theta'^{t-1} + \frac{L_{lv}}{c_{pd} \Pi} (C_{v \mapsto c} - E_{c,r \mapsto v}), \tag{2.23}$$

where on the right-hand side the first term and second term are the same as in Equation (2.13) and the third term is the non-adiabatic heating or cooling due to phase change.

Radiative Energy Flux Component

The third set of equations models how solar radiative flux is estimated, partitioned, and interacts with the surface and atmosphere causing a change in temperature as well as additional weather dynamics. Our radiation model is based on Stull [69] but adapted to urban scale and with a few additional simplifications to reduce computational cost. The predominating shortwave incoming solar radiation passes through the atmosphere, a part is reflected or absorbed by clouds, greenhouse gasses, and particles, and the rest is transmitted to the surface. On reaching the Earths surface the

radiative energy is partitioned into components that are used for water vapor/liquid phase change or latent heating; another part is used towards sensible heating and storage within the surface (Section 2.6.3). The outgoing radiation is predominantly longwave following Weins Law. The radiative flux heats the surface non-uniformly as a function of longitude and mostly the latitude of the simulated area. At a more local scale, the land scape and urban areas are significantly warmer (especially at night) than surrounding rural areas. The main factor behind such urban heat islands (UHI) is the distribution of land use as well as the use of man-made materials that effectively store long-wave radiation [70].

We use a force-restore method to compute the impact of the radiation on ground temperature, and the transfer of temperature from ground to air.

The objective of our radiation model is to compute a spatially and temporally changing value for the temperature T of each of the bottommost grid cells we refer to such temperatures as T_z. The radiation model defines constants, variables, and equations that are effectively below the bottom layer of the grid. Note that at ground level, $\theta \cong T_z$ because pressure is p_0. Once T_z is computed, changes are propagated upwards by buoyancy, and the energy, mass, dynamics-based prognostic equations.

Radiation budget

The radiation flux corresponding to one ground-level grid cell is split into four parts of a two-stream model,

$$Q^* = K\uparrow + K\downarrow + I\uparrow + I\downarrow, \tag{2.24}$$

where the shortwave solar radiation is reflected ($K\uparrow$) and transmitted ($K\downarrow$) and the longwave radiation is emitted ($I\uparrow$) up and diffusively radiated down ($I\downarrow$).

Shortwave radiation. The visible light transmitted by the sun can be quantized by

$$K\downarrow = S_c T_K \sin(\Psi) \tag{2.25}$$

where the solar constant $S_c = -1.127 K m s^{-1}$ is the intensity of incoming solar radiation at the top of the atmosphere, and T_K is the radiation attenuated by the depth

of atmosphere it has to travel (e.g., at sunset the radiation has a longer path to reach the surface) and by the amount of clouds, aerosols and other absorption/reflection components within the atmospheric layer.

To compute T_K, the model discretizes clouds into three different heights as low-level cumulus, mid-level altu and high-level stratus (i.e., low 0-2km, medium 2-6km, and high ¿6km clouds; with the fraction at each height being σ_{C_L}, σ_{C_M}, and σ_{C_H}, respectively. To compute this in our model, for each bottom-level grid cell, we use a 3D ray marching method setting the origin of the ray as the center of the grid cell, the step to sample each vertical grid cell, and as direction the sun direction. We accumulate the result of a cloud transfer function of each sample for each of the three height ranges, yielding σ_{C_L}, σ_{C_M}, and σ_{C_H}. The contribution (or weight) by the amount of clouds at each of the three heights, and other weights are based on simplifications introduced in Stull [69]. Hence,

$$T_K = (0.6 + 0.2 \sin(\Psi))(1 - 0.4\sigma_{C_L})(1 - 0.7\sigma_{C_M})(1 - 0.4\sigma_{C_H}). \tag{2.26}$$

The solar elevation angle Ψ is determined by the longitude/latitude of the urban space, time of day, day, and year:

$$\sin(\Psi) = sin(g_{lat}) \sin(d_s) - cos(g_{lat}) \cos(\delta_s) \cos[(\pi t_{UTC})/12 - g_{long}] \tag{2.27}$$

where g_{lat} and g_{long} are geographic latitude and longitude of the middle of the simulated region, t_{UTC} is the time in UTC, and δ_s is the solar declination angle:

$$\delta_s = 0.409 \cos\left(\frac{2\pi(d - 173)}{365.25}\right) \tag{2.28}$$

where d is the Julian day in the simulated year (and 173 represents the summer solstice day and 365.25 the number of days in a year).

The reflected shortwave radiation $K\uparrow$ is the fraction of $K\downarrow$ that is reflected by the surface as defined by its albedo (a):

$$K\uparrow = -aK\downarrow . \tag{2.29}$$

Longwave radiation. Earth re-emits the solar energy as radiation in the form of longwave infrared rays. The net longwave radiation is based on StefanBoltzmanns equation $\sigma\epsilon T^4$. $I^* = I\uparrow + I\downarrow$ is modeled by the empirically determined equation [69]:

$$I^* = 0.08(1 - 0.1\sigma_{C_H} - 0.3\sigma_{C_M} - 0.1\sigma_{C_L}). \tag{2.30}$$

Force-Restore Slap Model

Figure 2.13 pictorially represents the multiple temperature layers and the model variables of the force-restore method [71, 72]. As stated, the surface energy balance is performed by splitting the radiation (Q^*) into sensible heat flux (Q_H), latent heat flux (Q_L) and ground heat flux (Q_G). The division of surface energy defines four layers in the radiation model:

- Two atmospheric layers consist of (a) the bottom of the model grid with temperature T_z and (b) a layer near the surface with temperature T_a.
- Two ground layers consist of (c) a shallow layer of d_s cm thickness and temperature T_g where most of the soil temperature changes occur because of the suns radiation and thermal diffusivity (d) a deeper thick layer with seasonally varying mean temperature T_m. The depth d_s can be computed as

  $$d_s = \sqrt{v_g P/4\pi} \tag{2.31}$$

  where v_g is the thermal diffusivity of the soil and P is the period of simulation in seconds.

To compute the temperature T_z at height z, we start using a diurnal empiric *environmental lapse rate* value γ (which is true slope of variation of temperature with height) and the temperature of air closest to the surface, T_a. The empirical formulation of γ is given by

$$\gamma = 1.07 \cdot 10^{-8} t_l^5 + 8.18 \cdot 10^{-7} t_l^4 - 6.05 \cdot 10^{-5} t_l^3 + 7.72 \cdot 10^{-4} t_l^2 + 1.4 \cdot 10^{-3} t_l - 0.0184. \tag{2.32}$$

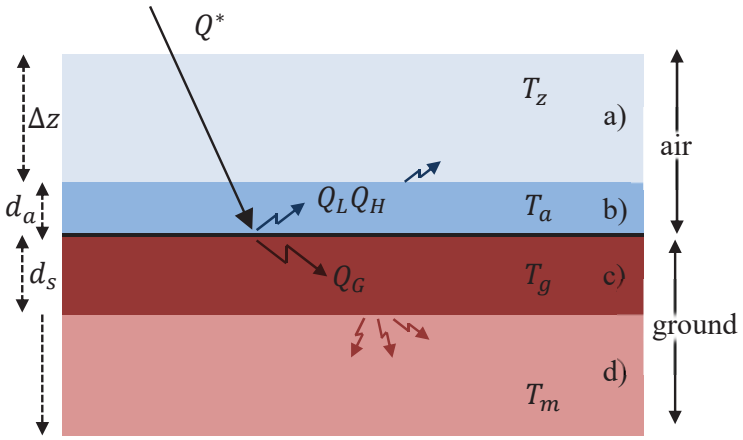

Fig. 2.13. **Force-Restore Slab Model.** Radiation gets to the surface and this heats T_g; a part gets diffused to deeper layers of the ground T_m, and a part heats the first few cm of air T_a, then T_a heats the bottom layer of our grid T_z.

Thus, T_z is computed as

$$T_z(z) = T_a + \gamma z. \tag{2.33}$$

In order to compute T_a, we first compute the ground heat flux

$$-Q_G = C_{GA}\frac{\partial T_g}{\partial t} + 2\pi\frac{C_{GA}}{P}(T_g - T_m) \tag{2.34}$$

where C_{GA} is the soil heat capacity per unit area computed as

$$C_{GA} = C_g d_s \tag{2.35}$$

and C_g is the soil heat capacity per unit volume (in $Jm^{-3}K^{-1}$) and it is a function of soil types and is available in standard textbooks (Noilhan Planito 1989).

To compute the change of T_g, we make use of the aforementioned radiation budget Q^* and the heat fluxes out of its layer interfaces as

$$\frac{\partial T_g}{\partial t} = \frac{-Q^*}{C_{GA}} + \frac{2\pi}{P}(T_m - T_g) - a_{FR}(T_g - T_a). \tag{2.36}$$

On the right-hand side, the first term is the force term quantifying the amount of radiation that reaches the surfaces and is transformed into heat depending on the soil

heat capacity C_{GA}; the second term is the restoration term measuring the conduction of temperature T_g to the deeper layer of temperature T_m; the third term represents thermal convection of the ground layer of temperature T_g to the first layer of air with temperature T_a, where a_{FR} is the conductivity between the ground and the air (note: $a_{FR} = 3 \cdot 10^{-4}$ when $T_g > T_a$ and $a_{FR} = 1.1 \cdot 10^{-4}$ when $T_g \leq T_a$).

Having computed the radiation Q^* and the ground heat flux Q_G, the sensible heat flux is computed by radiative balance. Instead of computing Q_H and Q_L directly, we use the ratio of sensible heat flux to latent heat flux, called Bowens ratio β that depends on the land use to compute Q_H directly:

$$Q_H = (-Q^* + Q_g)\frac{1}{1+\beta}. \tag{2.37}$$

Next, we estimate the change in T_a using a linear temporal function of sensible heat flux. Assuming heating from the surface is the sole source of increasing or decreasing air temperature and a constant boundary layer height (e.g., $z_i \approx 1km$ at noon and $z_i \approx 200m$ at night), we compute it as:

$$\frac{\partial T_a}{\partial t} = z_i^{-1} Q_H. \tag{2.38}$$

Numerical Stability

To improve the numerical stability and computational robustness during the simulation, we make several optimizations for time and space integration. For space integration, we use second-order central differentiation with the aforementioned Arakawa C-Grid. For time integration, we use Leapfrog integration and Robert-Asselin time filter [62]. Leapfrog numerical techniques, among other benefits, enables larger time steps (which leads to a faster runtime). Robert-Asselin time filter increases stability by updating any simulation variable ϕ at step n in the following fashion:

$$\bar{\phi}^n = \phi^n + 0.1(\phi^{n+1} - 2\phi^n + \bar{\phi}^{n-1}). \tag{2.39}$$

Intuitively, using the former time step filters changes over time and creates a more smoothly varying solution. Combining Leapfrog integration and Robert-Asselin filter-

ing (i.e., using $\bar{\phi}^n$) increases the maximum time step s to approximately four seconds. From Durran [62], Leapfrog and Robert-Asselin are stable for $s = 1$ second. Therefore, it can be calculated that it will be stable if

$$\Delta t \left(\frac{u_{max}}{\Delta x} + \frac{v_{max}}{\Delta y} + \frac{w_{max}}{\Delta z} \right) \leq 1. \tag{2.40}$$

Giving that $u_{max}, v_{max} < 50m/s$ in our simulation (i.e., higher winds are only found in tornados) and $w_{max} < 30m/s$, the maximum time step for $\Delta x = \Delta y = 1000m$ and $\Delta z \geq 200m$ is $\Delta t \leq 4$ seconds. The model is tested for stability through the CFL criteria in terms of the time step, variables, and grid spacing constraints.

Boundary Conditions

For top and bottom boundaries, we set zero wind and moisture variables. Potential temperature, Exner function, and pressure are defined as the boundary value stretched out through the vertical domain:

$$\begin{cases} \phi(z < 0) = \phi(z \geq grid_z) = 0 & \text{for } \phi = \{u, v, w, q_*\} \\ \phi(z < 0) = \phi(z = 0) & \text{for } \phi = \{\theta, \Pi, \rho\} \\ \phi(z \geq grid_z) = \phi(z = (grid_z - 1)) & \text{for } \phi = \{\theta, \Pi, \rho\}. \end{cases} \tag{2.41}$$

2.6.4 Weather Forward Design

The user wants to quickly design a city model and simulate realistic weather for any length of time (e.g., days, months, or even years). This approach is useful to virtual environments, modern games, education, and decision tools where the goal is realism by taking advantage of the detailed amplification of procedural modeling. The user defines the desired input parameters.

Usually, this means drawing the land use distribution (ω_l), define the urban procedural parameters (ω_p), and choosing some initial weather conditions (ω_w). Given these initial conditions, our weather simulator is then executed for the desired time

period. Note that the simulation varies from one to the next day because i) the initial conditions are altered through the former day, ii) the radiation model that directly affect the wind and buoyancy changes with the day of the year due to the sun trajectory, iii) the potential of rain also changes depending on the season.

2.7 Summary

In this chapter, we have presented our forward procedural engine to generate 3D urban models. We have also presented our traffic and weather simulators and how we can couple them with the urban model to create a forward design tool for the user.

3. INVERSE DESIGN

Inverse Procedural Modeling is a set of techniques that provides control to the user over the output procedural models, i.e., tries to solve the limitations of procedural modeling (Chapter 2). This is achieved adding an extra layer to the procedural model to automatically infer the input parameters or rules to generate the desired procedural model. *Inverse Design* includes inverse procedural modeling but extends the idea to any system that has a set of input values and has as output a set of indicators and output values. The system is represented as a 'black box', such that the user does not have to know how internally works. We use this idea to control the procedural generation of urban models as well as to control the simulation of traffic and weather.

We propose coupling a forward procedural urban modeling process with an inverse design technique that optimizes the input parameters to satisfy user-specified goals. We are closing the loop between forward and inverse modeling strategies (as depicted in Figure 3.2) and provide both increased control and higher flexibility, enabling the user to be much more efficient in generating a model that satisfies their requirements. Both forward and inverse modeling strategies can be applied during an interactive editing session for either local or global model modifications. The key advantage of supporting an inverse modeling methodology is that the procedural model and simulation can be controlled in new ways without having to re-program the forward engine or change the simulator. Our work abstracts the urban procedural model into a general parameterized form and adds a component that is able to discover how to control the procedural model so as to obtain the desired values for user-specified indicator values derived from the resulting model.

3.1 Related Work

In recent years, there has been an increasing interest in this area. Šťava et al. [13] presented an innovative procedural modeling of rules and parameter values. The authors focus on L-Systems on a 2D content. The terminal symbols are known, and they generate context-free rules for linear structures. Bokeloh et al. [14] explored partial symmetries to complete the geometry of ill-specified input models exhibiting certain symmetries. They modify the model from shape rules extracted from the partial symmetries in the model. Lipp et al. [30] presented a method to control locally the 3D output of a procedural model to produce a desired model. They provide local editing, combining layouts using graph-cuts and providing persistence and layering. Beneš et al. [73] presented the idea of guided procedural models, where the model geometry is divided into guides. This approach allows to control the overall shape of the model by manipulating these guides.

Park et al. [74] computes a grammar from animated sequences to provide a tool to generate new animations. Aliaga et al. [15] created a grammar from a manually segmented building polygons to model new ones. Other works have focused on individual facades [17, 18] and furniture layouts [75, 76], they proposed methods to determine procedural parameter values.

More close to our work, Talton et al. [77] described a enhanced MCMC Metropolis-Hasting approach for learning a varying set of parameters of a procedural model so as to obtain an output following a desired global shape. The results were very interesting but not fast enough for interactive modeling tools (e.g., a multi-building example takes about 14 min) and requires the grammar to be context-free. Šťava et al. [78] presented an MCMC-based optimization method that use 3D polygons of trees as input to estimate the parameters of an L-System so that to produce procedural models similar to the inputs. Recently, Ritchie2015 et al. [79] presented a method to control a procedural model using a stochastically-ordered sequential Monte Carlo method (SOSMC). In their system, each thread is executed with an independent

order and use it to control the shape of the final model. In summary, unlike previous systems using MCMC within the context of architectural or procedural modeling (e.g., [75–77]), i) we support complex indicators, thus enabling control beyond global shape, such as by high-level semantics and indicators, ii) we consider the procedural model as a black box and thus support context-free and context-sensitive stochastic grammars, and simulators, and iii) our system is interactive and able to alter models to control and design new models.

3.2 Our Inverse Procedural Model

Figure 3.1 presents the outline of our forward and inverse design. In inverse design, the user defines a set of target indicators to control the model. Depending on the context and the purpose, the indicators can be as simple as a single value (e.g., temperature at the city center, distance to park) to complex outputs with semantic meaning (e.g., landmark visibility, rain intensity). Our MH method efficiently searches the high dimensional space to find such solutions. Altering the input parameters, the optimization is able to find the necessary values to generate the desired model. We call this process *inverse design*. Users, unaware of the rules of the underlying urban procedural model, can alternatively specify arbitrary target indicators to control the modeling process. The system itself will discover how to alter the parameters of the urban procedural model so as to produce the desired 3D output. We label this process inverse design.

Since the solution space is large and the system is non-linear, our method is based on Markov Chain Monte Carlo (*MCMC*), more specifically on Metropolis-Hastings (*MH*) method. MCMC is a group of stochastic methods that sample probability distributions based on a Markov chain. We use MH to randomly walk the domain space using a probability distribution and different energy levels in order to find a solution.

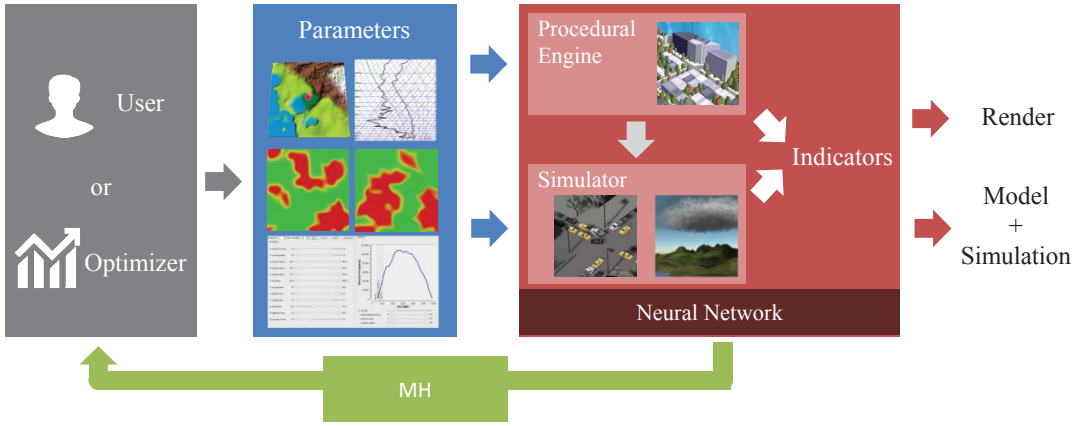

Fig. 3.1. **Our Inverse Design.** The user or a optimization tool controls the input parameters to create a city and optionally simulate to compute some output values or indicators. Using an optimization method, Metropolis Hasting, we can find the desired behavior or model.

Depending on the design and purpose, our MH based method will have different variations:

- **Objective optimization:** The user defines some high-level indicators, and the system optimize a set of parameters to find those that yield the desired behavior. For example, the user defines the cloud coverage throughout a day, the optimization runs and finds the initial conditions necessary for such weather behavior.

- **Cost minimization:** The user defines an indicator goal but, at the same time, wants to find the solution that is as similar as possible to the original model. For instance, we want to reduce the traffic in the center of a city, we could alter all the road network, however, we can optimize the traffic making the minimum changes to the road network.

- **Constraint optimization:** The user can define extra constraints to the domain space as well as the number of changes made to the parameters. For

instance, the user defines that the possible change for the optimization is the land use distribution, but the user defines that the distribution can just be altered in a 10%.

Using these three methods, we get control on the design and allow us to create a wide variety of results for urban procedural models as well as to control traffic and weather.

3.2.1 MCMC-based Approach

The user wants to find a 3D model that exhibits a set of target indicator values $\Gamma^* = \{\bar{\gamma}^*, \check{\gamma}^*\}$ where $\bar{\gamma}^*$ is the mean and $\check{\gamma}^*$ its standard deviation. Our system runs the inverse design (MCMC-based optimization) to interactively find a set of input parameters Ω that better satisfies the user-specified target indicator values.

We use an MCMC-based approach, more specifically based on the Metropolis-Hasting (MH) algorithm [80], [81], to find a set of input parameter values Ω that generates a 3D model such that the computed indicator values are a good approximation to the user-specified indicator values Γ^*. Our method proposes new states Ω_{t+1} and tries to minimize the current measured indicator values (Γ) with respect to the user-specified one such as $|\Gamma^* - \Gamma| \rightarrow 0$.

3.2.2 Seeding Initialization

Our method starts from one or more initial parameter values or *seeds* Ω_0. The user can choose whether to use a single seed consisting of the parameter values of the original model or multiple seeds.

In general, the initial seeds are chosen uniformly and separately sampling each parameter within its extents $(\omega_{min}, \omega_{max})$. If the user defined the desired range of target input parameters, they are also sampled to generate the initial seeds. Our system uses n_β different temperatures, with n_p initial seeds, for n_i.

3.2.3 Search Process

Given the current state Ω_t, a new candidate state change Ω_{t+1} is proposed by adding to each parameter ω a value sampled from a Gaussian distribution $\mathcal{N}(0, \alpha|\omega_{max} - \omega_{min}|)$ where α is the perturbation change, typically set to 5%. Since the distribution of sampling is symmetric (mean 0), it follows the Metropolis-Hasting algorithm and the acceptance ration can be computed as

$$a(\Omega_t \to \Omega_{t+1}) = \min\left(\frac{p(\Omega_{t+1})}{p(\Omega_t)}\right) = \min\left(1, \frac{\exp\left(-\beta E\left(\Omega_{t+1}\right)\right)}{\exp\left(-\beta E\left(\Omega_t\right)\right)}\right)$$

where β is the chain's temperature, when β is small is more likely that candidates with bigger error will be accepted, in contrast, when β is big, it will be more likely that just improvement candidates will be selected. This helps to avoid local minima and find better solutions.

3.2.4 Acceptance Ratio

Once a candidate state Ω_{t+1} has been sampled, we run the procedural engine and/or re-simulate and use a modified Metropolis ratio to compute the probability to accept this new candidate state. Note that if the candidate state Ω_{t+1} is not accepted, the transition $\Omega_t \to \Omega_{t+1}$ does not occur and the next state is Ω_t again.

A standard Metropolis-Hasting acceptance ratio when the proposed sampling distribution is symmetric is defined as

$$a_{min}(\Omega_t \to \Omega_{t+1}) = \min\left\{e^{-\beta[E(\Omega_{t+1}) - E(\Omega_t)]}, 1.0\right\}. \tag{3.1}$$

3.2.5 Error Function

The function to minimize is $|\Gamma^* - \Gamma| \to 0$, however, we just can alter the input parameter Γ to find the solution. Therefore, we define the error function as

$$E\left(\Omega_t, \Gamma^*\right) = \frac{1}{n}\sum_{\gamma \in \Gamma}\left(\frac{|\gamma - \gamma^*|}{\check{\gamma}^*(\gamma_{max} - \gamma_{min})}\right)$$

Optionally, the user can add an additional term to the error function such that the solution is close the preferred input parameter values.

3.2.6 Indicators

To illustrate our inverse design, we have implemented several indicator values. For urban modeling, we have implemented: floor-to-area ratio, distance to park, visibility, sun exposure, and interior light. For traffic design, we have design traffic zone and emissions indicator. For weather design, we have implemented: rain intensity, cloud coverage, and temperature.

In the rest of this chapter, we will describe the specific behavior and sampling for urban models, traffic, and weather simulators.

3.3 Inverse Design of Urban Procedural Models

Urban procedural modeling is becoming increasingly popular in computer graphics and urban planning applications. A key basis for the popularity of city-scale urban procedural modeling is that once the procedural model is defined, it encapsulates the complex interdependencies within realistic urban spaces [82] and enables users, who need not be aware of the internal details of the procedural model, to quickly create large complex 3D city models (e.g., [9–12]). Effectively, the detail amplification inherently provided by procedural modeling is exploited: a small set of succinct *input rules* and *input parameters* can yield very complex and coherent outputs. However, the succinctness of urban procedural modeling is also its Achilles' heel: obtaining a 3D urban model with complex user requirements is a challenging task that requires experience and in-depth knowledge of the underlying procedural model. An expert user with programming skill must set the values for the input parameters, implement the procedural rules in software, and iterate between code, parameters and examination of the output to achieve the desired model. In short, what is needed is a means to efficiently learn the parameters and rules required to produce a desired 3D urban

model, without requiring the end user to write complex software programs. Urban planners and content designers often have a clear vision for the target urban model, but lack the computational tools to rapidly create models that meet their design requirements.

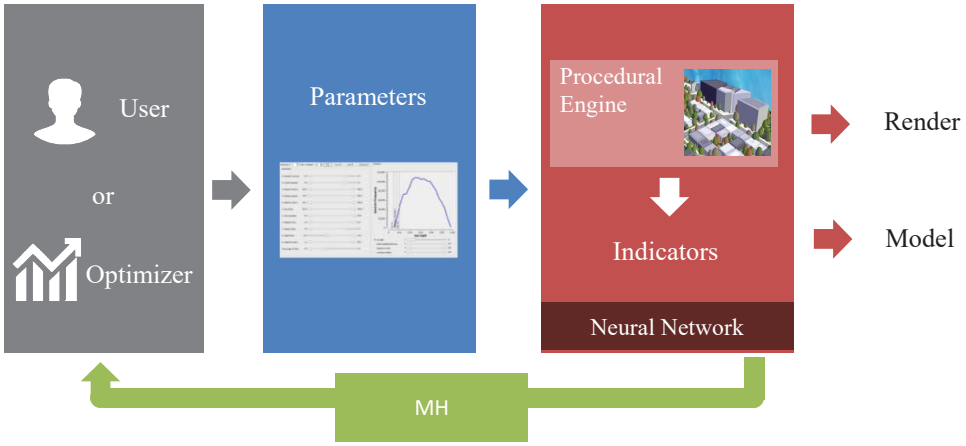

Fig. 3.2. **Urban Modeling Pipeline.** Visual representation of our design modeling pipeline of urban models.

To achieve interactive rates and still find the desired model that produces the target indicator, we use Artificial Neural Networks [83, 84] (ANN) to speed up the process. If it typically takes 250ms to generate one procedural instance and to compute the indicators, we would just be able to check very few instances to maintain interactivity. However, our inverse computation needs to run many chains with different temperatures for many steps ($O(10^4)$) to be able to find the solution. Instead, we train our ANN to learn how the target indicators behave for different input parameters. Then, our MCMC-based approach instead of having to explicitly produce a 3D model and compute the target indicators for each candidate state, we use our ANN to predict the behavior. Note that since not all target indicators are feasible, we additionally provide an analysis tool that informs the user if a particular indicator value is feasible.

Using our system, we can interactively modify 3D models for urban areas spanning over 100 km^2 and containing up to 10,000 parcels. Our typical frame rate while changing indicator values interactively and generating new 3D models is 2.5 to 10 frames per second. The back-propagation engine is able to compute indicator values that are typically within 5% of using the actual procedural model and require only a small fraction of the compute time (e.g., < 0.01 ms as opposed to a typical 250 ms using a procedural modeling engine, thus an effective speedup of over 25,000x). The user can choose from a proposed best set of 3D models. Subsequently, more edits can be made, the model can be saved to disk, or the geometry exported to commercial rendering engines.

3.3.1 Urban Search Approach

We present two modes to search: local and global moves. *Local moves* are as described in Section 3.2.3. Additionally, we use a precomputed frequency distribution of the indicators to do *global moves*. Global moves jump to areas where it is more likely to find the target indicator values as per previous executions. After the proposed state is generated, an acceptance ratio is evaluated. If the ratio is satisfied, the new state Ω_{t+1} is accepted and the next step start from it, otherwise, the transition does not occur and the following proposed step will be based on the old Ω_t instead of Ω_{t+1}.

Global Moves

Global moves are computed using a histogram of the frequency of the indicators. The parameter space is sampled homogeneously ($O(10^5)$) and the indicators computed. Then, for each indicator we compute a histogram with n_f bins that spans $(\omega_{min}, \omega_{max})$. When a global move is performed, a single indicator γ is selected randomly then with probability h_m the highest count bin of that indicator is selected and with $(1 - h_m)$ the bin that contains the target indicator is selected (Figure 3.4a). Finally, from the selected bin, one of the samples states is selected and used as Ω_{t+1}. Global moves

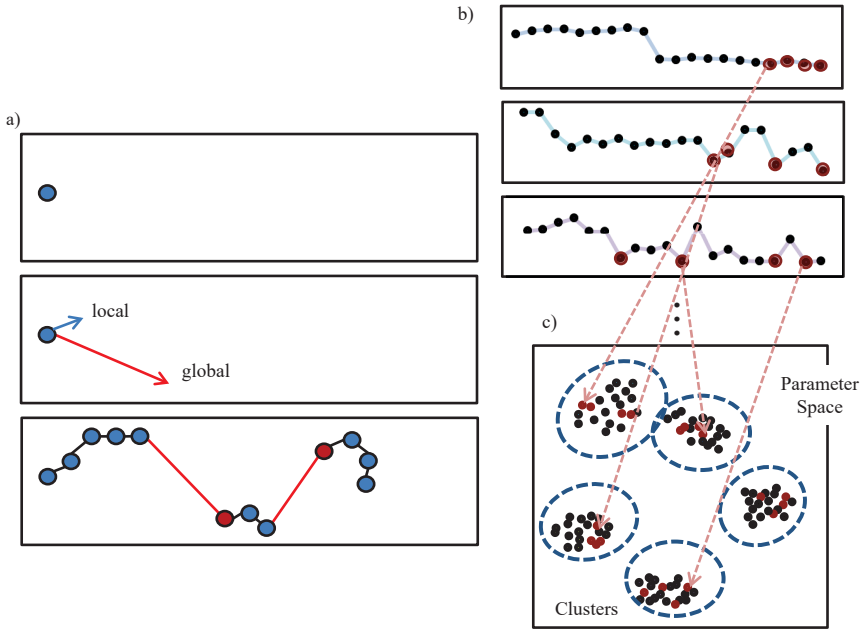

Fig. 3.3. **Parameter Searching Overview.** a) The chain start on a seed and attempts local or global movements. b) The process is repeated n_I steps and for different temperatures. The best solution are mapped to the parameters space. c) These solutions are clustered and one solution per group is shown to the user as possible solution.

tend to move to areas with likely values of the indicator and areas where the target indicator is.

Artificial Neural Networks

We have described that our MCMC based approach needs $O(10^4)$ samplings to find the desired solution. Since the complexity of the procedural engine and the indicator computation is high (e.g., 250-500ms computation time), it would not be feasible to achieve interactively and keep the same sampling rating. Reducing the sampling is not possible since the best solution found with few samplings would be very far from the user-user specified indicators. In contrast, we propose to use ANN [83,84] to

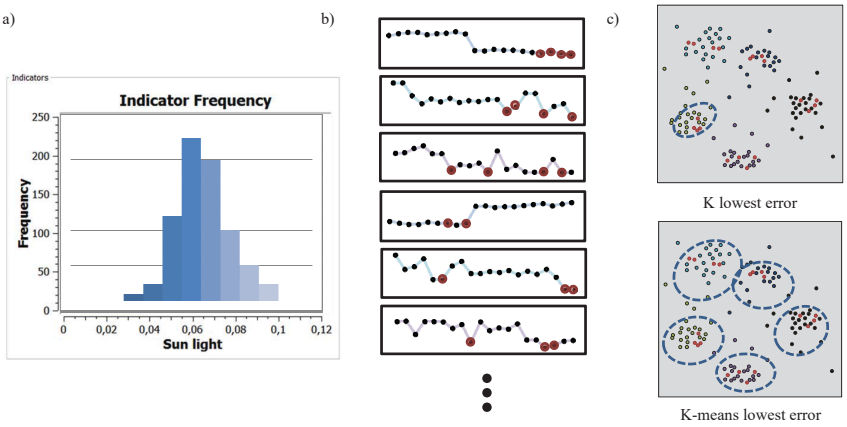

Fig. 3.4. **Indicator Histogram and Clustering.** a) Example of a histogram. b) n_β chains are run and the top 25 best solutions (lowest error) are selected. c) If we select the top k lowest error, the input parameters are very similar. d) In contrast, if we use $k-means$ clustering, we find solutions but with different input parameters that yield visually different models.

replace the procedural engine and indicator computation to achieve interactivity. This approach allows us avoid creating for each proposed state the scenario and indicator computation. Instead we use the ANN to predict the indicators directly from the input parameters in a fraction of a millisecond.

ANN is an adaptive system (Figure 3.5) that changes its structure (weights) based on a training set. Each neuron (Left Figure 3.5) has a set of inputs, one output, and a set of weights. The training process modifies the weights of each neuron such that for the given training inputs it generates the given training output. After the training is done, the ANN can predict new indicator values using the new input parameters. One neuron is not usually enough to learn the complexity of the system behavior. Therefore, more layers of neurons are added (hidden layers). The neurons are then interconnected to form a multi-layer perceptrons, i.e., a neural network (Right Figure 3.5). In our case, to define the number of hidden layers, we test different number of hidden layers to find the one with best accuracy.

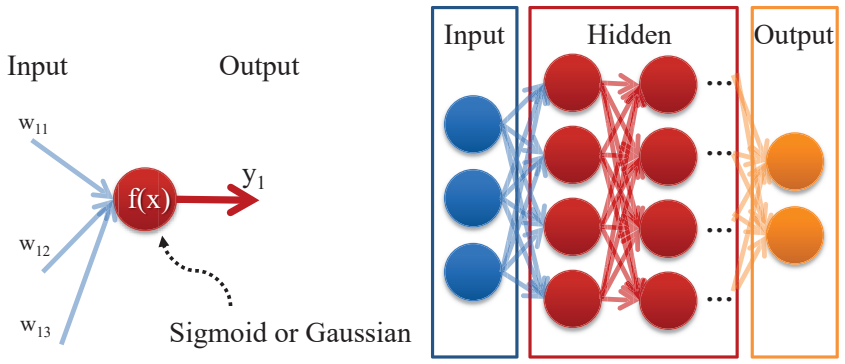

Fig. 3.5. **Artificial Neural Networks.** (Left) Example of a NN, the neuron receives a set of input w and produces an ouput y. The training process find the weights in the neuron/s function such that the desired output is generated when the training input is used. (Right) General structure of a ANN.

Our ANN training process wants to learn how to predict the output indicators from the set of input parameters. Since the indicator values behavior depends on the number and the distribution of place types, we need to train the neural network for each scenario. In practice, static indicators (i.e., those that do not depend directly on the existence of different place-types) do not require any re-training. Local indicators (e.g., landmark visibility) do need to be retrained when major city changes are made. Nevertheless, in all of our shown examples, we train the neural network only once. The ANN training receives as input: i) the set of parameters that the user allows to change, in general, it will be $m = zm_p$ (i.e., number of place-types times the number of parameters), ii) their feasible ranges, iii) the distribution of place types over the target area, iv) the set of indicators that the user wants to control n. The number of necessary training samples depends on the target and the number of place-types. However, we found empirically that using 200-500 samples was enough. Finally, we optimize the number of hidden layers running different configurations around $(m - n - 1)$ to achieve the best accuracy.

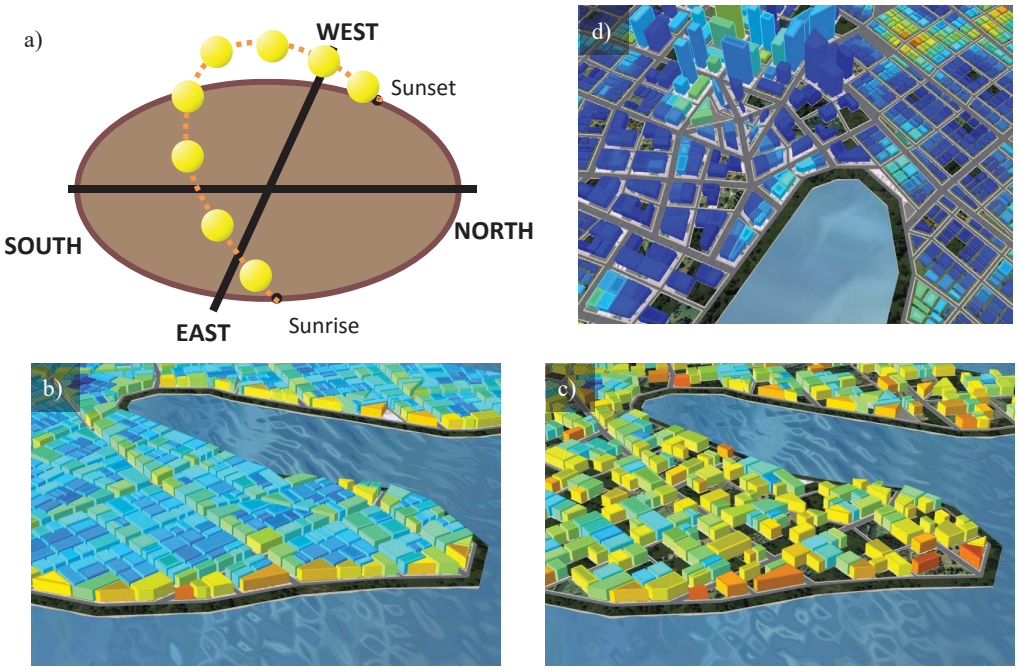

Fig. 3.6. **Indicators.** a-c) Sun exposure. a) We compute the sunlight per facade as the percentage of sun that reaches the building facade averaged during the course of a day, over all days of the year. b-c) shows two different models for different levels of sun exposure. d) shows with a color coded schema the distance to park (blue means close to park, red means far to park)..

After the training phase is complete, the ANN is ready to predict the output indicators just using the values of the input parameters. The parameter input values are feed to the first layer, then the values retrieved from the previous layer are transformed using the function weighted by the training until reach the last layer, those values are the predicted indicator values. This approach avoids the need to compute the 3D model and indicators, increasing the performance 25000x.

3.3.2 Urban Indicators

The implemented indicator values (Figure 3.6) are:

- **Floor-to-Area Ratio:** It is the total area of the building divided by the total area of the footprint/lot the building is located on.
- **Distance to Park:** Distance from a parcel center to the closest park.
- **Landmark Visibility:** Percentage of buildings that can see a designated building or landmark. It is consider that it is seen if there are not complete occlusion from the current building to the landmark.
- **Sun Exposure:** We compute the sun exposure as the percentage of sun that reaches the building facade averaged during the course of a day, over all days of the year.
- **Interior Light:** The natural interior light ratio is the sun sun exposure of the facades divided by the average minimum distance from an interior building point to a facade.

Since each indicator is calculated for each place-type, each scenario counts with $n = zn_t$ total indicator values where $n_t = 7$ is the number of indicators per place-type.

3.3.3 Urban Seeding Initialization

We use the seed initialization described in Section 3.2.2 and we set the values experimentally to $n_\beta = 4$, $n_p = 2$, and $n_i = 5000$. Local or global move with probability q_l and $(1 - q_l)$, respectively.

3.3.4 Urban Solution Selection and Feasibility

Figure 3.4b-c shows an overview of the solution selection. Our method explores the domain space through n_β temperatures, n_P initial seeds, and n_I moves (Figure 3.4b). For each state, the indicator values are computed and compared with the user-specified values. If the user wants to see k different alternative solutions, we could select the k best solutions (Figure 3.4c top). With this approach, it is very

likely that those solutions have very similar input parameters, yielding very similar models. In contrast, if we perform $k-means$ algorithm to the top 25 best solutions per chain, we can find solutions close to the user-specified indicator values but with different input parameters.

Note that not all target indicator values are possible. For this task we provide an interactive tool based on the global moves histograms 3.3.1. Each indicator variable histogram contains the distribution of feasible indicator values, if the target indicator values are outside this range, that indicator cannot be achieved for that conditions.

3.4 Inverse Design of Traffic

Traffic is well recognized to be difficult to simulate and control due to its highly nonlinear behavior, inherent complexity, and emergent behavior. For example, a local change to the network might have an adverse effect elsewhere in the network (e.g., blocking an important avenue in one neighborhood might cause a long traffic delay in another part of the city due to traffic redirection, or increasing the speed limit of a road segment might, in fact, seem like it will improve travel time, but it might in fact attract more vehicles and ultimately slow down transit in the area). Trial-and-error and keyframe-based control techniques might work for a small number of intersections but not for interactively designing large-scale traffic animations. Therefore, to create an inverse traffic tool, we face the following challenges i) simulating realistic traffic flows at interactive rates, and ii) controlling traffic in an easy and intuitive manner.

Our methodology automatically creates a 3D urban model (e.g., an interconnected network of roads, parcels, and buildings) that exhibits a desired and realistic vehicular traffic behavior (Figure 3.7). Our method provides an interactive virtual paintbrush tool whereby the user can specify i) the desired traffic for a new urban model (e.g., for games and films, navigation services, or virtual environments in general), or ii) can improve or alter traffic in a provided urban model to assist traffic planners in obtaining desired values for standard metrics such as road occupancy (i.e., percentage

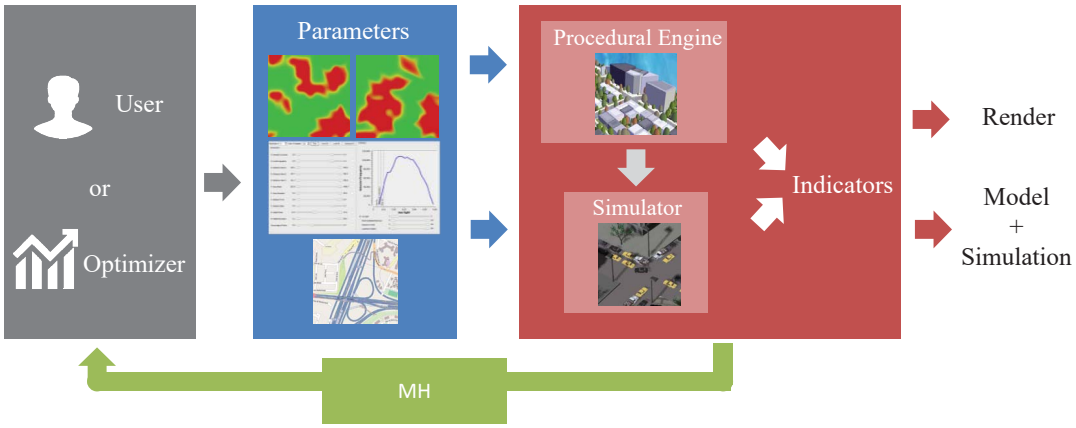

Fig. 3.7. **Traffic Pipeline.** Our approach enables a designer to specify a vehicular traffic behavior and the system will compute what realistic 3D urban model yields that behavior. The user defines the job and people distribution and defines the procedural parameters or load a road network; inputs are used to simulate traffic and the user draws a desired new traffic behavior (or traffic optimization). Our system iteratively simulates and alters the model so as to find solutions that meet the desired goals and/or costs.

of the segment that is occupied), travel time, or CO emission level. Our solution includes a novel traffic microsimulation engine and an algorithm to manipulate traffic behavior. To the best of our knowledge, our framework is the *first interactive method to automatically generate a realistic 3D urban model that yields a specified traffic behavior.*

Our traffic microsimulation engine yields both the detailed per-vehicle data needed for traffic animation and the fast performance needed for our design strategy. The system we create achieves its significant speedup by extending microsimulation with a novel traffic atlas concept, a new approximate solution to a time-dependent shortest path problem, and an efficient adaptation of car-following, lane changing, and gap-acceptance models.

Our traffic manipulation strategy explores which set of urban model changes brings the simulated traffic behavior closer to the interactively specified behavior.

3.4.1 Previous Work: Traffic Design and Animation

Network (re)design to satisfy a set of traffic objectives is studied in the transportation community (e.g., [85]). Often solutions are formulated as bi-level leader-follower games typically known as Stackelberg games [86]. However, it is well known that these types of problems are NP-hard. Approximations are typically analytical optimizations at non real-time speeds. Recently, within the computer graphics community, Go et al. [31] animate vehicles using control and motion planning. Van den Berg et al. [32] reconstruct and visualize continuous traffic flows from discrete data provided by traffic sensors. Sewall et al. [33] extend a continuum based (i.e., macroscopic) approach to including some lane and speed limit changes. Sewall et al. [2] extend the method to include microsimulation in selected parts of the road network while obtaining roughly similar performance. Similar to Subsection 2.5.2, all these methods are also forward-generating. In contrast, our method supports our novel traffic design concept. Further, Sewall et al. [33] claims a performance of about 100x over real time; our method yields a microsimulation at 9000x improvement over real-time.

3.4.2 Traffic MCMC-based Approach

We describe our traffic manipulation methodology and its use of the traffic simulation engine. During our MCMC process, the probability of a road network change is computed as a product of a change-type probability and a location probability. Each proposed change is evaluated for acceptance and the search continues until satisfying an objective function (or reaching a maximum "cost").

When so desired, the aforementioned traffic painting can be used to constrain areas of the road network and ensure only local road network changes. However, due to the global and complex nature of traffic flow, a closer to the optimum solution for a locally specified objective might involve a global set of changes – thus stability can be enforced but is not always beneficial.

Change-Type Probability

The following topology-preserving and topology-changing operations worked well with our examples. For a lane i or person/job grid cell k, we define four change types and their probabilities.

- *Lane direction change.* With probability d_i this topology-preserving change alters the directionality of a road segment's lane (Figure 3.8). To decrease traffic in a zone, we switch the lane direction to direct vehicles away from the zone middle. Conversely, to increase traffic, we swap the lane direction so that vehicles pour into the zone.

- *Number of lanes change.* With probability n_i this topology-preserving change alters the number of lanes per direction of a road segment. To decrease traffic in the zone, a lane is added to a road segment inside or outgoing from the traffic zone. Conversely, to increase traffic a lane is removed from a road segment.

- *People change.* With probability p_k this topology-changing operation relocates the people within the urban space and potentially triggers a significantly different procedural urban model (Figure 3.9). To reduce traffic, we need to know the cells that generated people who passed through the traffic zone and then relocate their distribution energy to elsewhere in the urban space. To increase traffic, we move the distribution energy from a random area to the most common people distribution area with vehicles passing through the traffic zone. Whenever possible, we transfer people from a zone wishing to decrease traffic to another zone seeking to increase traffic.

- *Job change.* With probability j_k this topology-changing operation is analogous to the previous category but moves job distributions instead.

These four types of changes may alter the cityscape. In particular, people/job distribution may alter buildings (e.g., dense areas will have smaller setbacks and

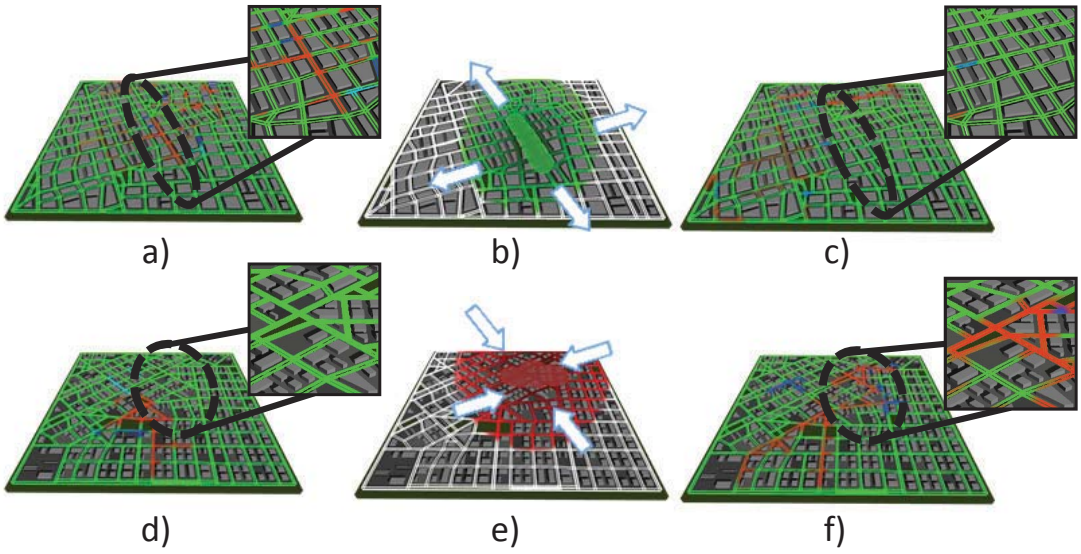

Fig. 3.8. **Lane-Changing Operations.** a) Initial network occupancy. b) The user "paints green" so as to reduce traffic. c) After MCMC, new road occupancy values closely match the painted traffic behavior. d-f) An analogous process but for increasing lane traffic.

bigger buildings) and the road network (e.g., MCMC adds roads to fill a dense area – even to OSM networks).

To avoid excessively altering lane directions/number of lanes, changes are done to a neighborhood of lanes. Further, to find origin-destination link pairs that pass through the traffic zone for people/jobs changes, we create a hash table to lookup the vehicles used by each edge. Then, we create a histogram of origin-destination link pairs of those roads and find which were the most common edges that made the vehicles cross the traffic zone.

Location Probability

The probability of changing all lanes, people, and jobs inside a zone is related to their location relative to the zone. The location probability x_i of a lane changing

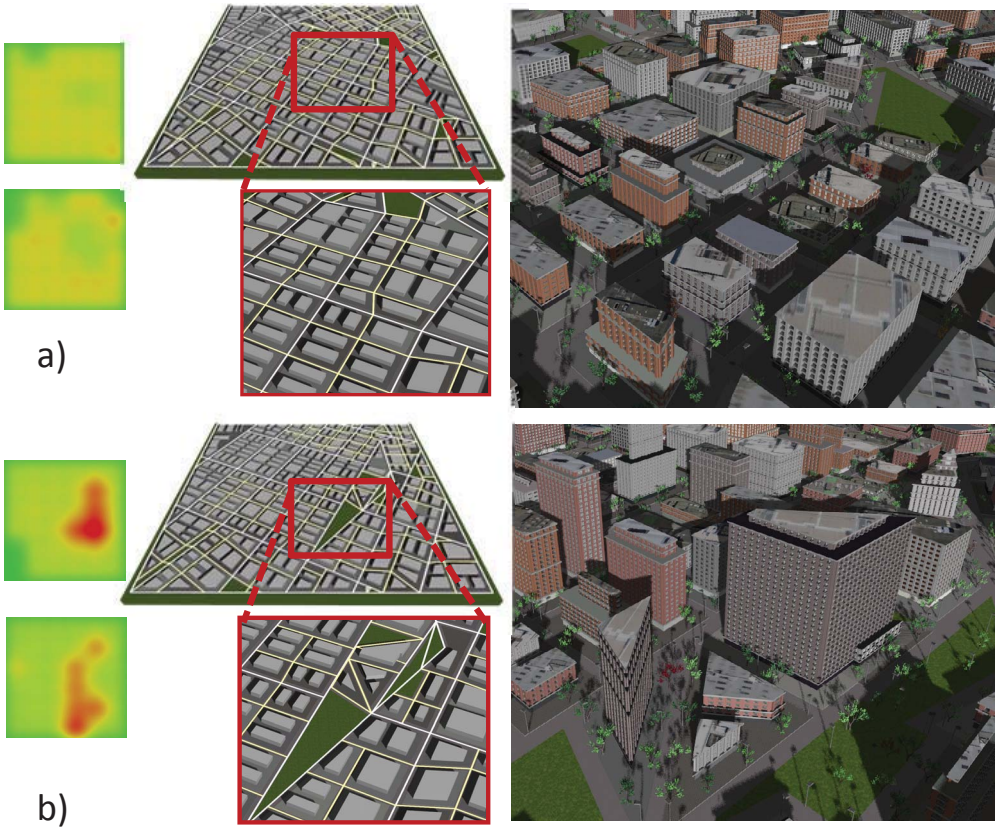

Fig. 3.9. **People/Jobs Changes.** Such changes can impact network topology. a) Input city. b) City with changed roads and buildings satisfying new traffic.

direction, or the corresponding road segment having a number of lanes change, is inversely proportional to the distance from the lane's midpoint to the exterior border of the traffic zone. Our function is setup so that a lane inside a zone has probability one and those outside the zone are computed using a Gaussian weighting function; i.e.,

$$x_i = \begin{cases} \frac{1}{w_Z}\phi\frac{t_i}{w_Z} & if\ outside\ zone \\ 1 & if\ inside\ zone \end{cases} \tag{3.2}$$

where t_i is the distance of lane i to the zone perimeter and w_Z is the width of a zone's area of influence on the surrounding urban space. In our system, t_i is computed as number of graph edges links (e.g., breadth-first search).

The location probability q_k of a people change, or job change, is proportional to its usage as an origin, or destination, for a route passing through the traffic zone. Thus, its location probability does not depend on whether the cell is near the traffic zone but on whether the cell is used by a route that passes through, or near, the traffic zone. The aforementioned histogram of origin-destination pairs is sorted and top candidates are given the highest probability. Thus,

$$q_k = \begin{cases} \frac{1}{w_S}\phi\frac{s_k}{w_Z} & if\ outside\ zone \\ 1 & if\ inside\ zone \end{cases} \tag{3.3}$$

where s_k is the histogram count for cell k and w_S is a normalization constant. To instill some additional randomness in the solution exploration process for people and job changes, we add an additional probability r. The term r mimics behaviors in a city where people, or jobs, periodically change due to a variety of reasons. In predictive agent-based urban simulations (e.g., [87]), such periodic random change of people or jobs is modeled by allowing a small percentage (e.g., 10%) of changes per year.

3.4.3 Traffic Seeding Initialization

To improve performance when the desired traffic behavior is significantly different from the current one, we make an initial guess of the desired distribution of people and jobs. This task is related to the networking problem of computing a traffic matrix which specifies the amount of traffic between source-destination pairs (i.e., person-job pairs). We adapt the tomo-gravity model [88] which is one such well-known method to infer the traffic matrix from the link loads (i.e., road segments). Once such initial locations are computed, we use MCMC optimization to refine the result.

Each thread starts using a different random seed and a different temperature β with values that were empirically found to range from 4 to 256.

3.4.4 Traffic Search Strategies

Our optimization process includes two objective functions: goal-driven and cost-driven.

A candidate state change Ω_{t+1} is obtained by sampling over a subset of the possible lanes. The probability is equal to the product of the corresponding change-type probability and location probability: for each lane $d_i x_i$ or $n_i x_i$; whereas for each grid cell $p_k(q_k + r)$ or $j_k(q_k + r)$. Starting with the lanes nearest to or inside a zone and the people/job cells highest in the origin-destination histogram, we randomly select lane direction changes, number of lanes changes, people relocation, and/or jobs relocation. We grow the subset until H changes occur amongst all change types. The collection of lanes, people, and jobs changes define a candidate state change Ω_{t+1}.

We use the acceptance ratio as described in Section 3.2.4. Once $c = n$, we have reached the final state and the best solution over all threads and states is selected.

Goal-Driven Optimization

Our goal-driven optimization drives the simulated traffic behavior to the road occupancy specified by the traffic zone set. Road segment s's occupancy is defined as

$$u_s = \frac{c_s}{L_s n_s / b_v} \tag{3.4}$$

where c_s is the average of the number of vehicles on the road segment over the simulated period and the denominator is the maximum number of vehicles per road segment (also known as the "jam density"). The maximum vehicles per road segment is computed using L_s= length of the road segment, n_s= number of lanes in the road segment, and b_v= minimum distance between vehicles (e.g., 5m). The objective function can be then written

$$F(X_n, \{Z_1, Z_2, ..., Z_K\}) = \sum_S \|u_s - \widehat{u}_s\| \tag{3.5}$$

where S is the set of roads inside the traffic zones and $\widehat{u}_s$ is the desired per road segment occupancy value.

Cost-Driven Optimization

Our second strategy minimizes the cost of the network changes once traffic behavior is sufficiently similar to the one specified by the traffic zones. For traffic design, this enables finding a low-cost solution satisfying the behavior. To measure cost C, we quantify the cost of all changes:

$$\begin{aligned} C = (w_L/N_L) \sum_{N_L} \|e_i - \bar{e_i}\| + (w_S/N_S) \sum_{N_S} \|n_s - \bar{n_s}\| + \\ (w_P/N_P) \sum_{N_P} \left\|P - \bar{P}\right\| + (w_J/N_J) \sum_{N_J} \left\|J - \bar{J}\right\| \end{aligned} \tag{3.6}$$

where (w_L, w_S, w_P, w_J) and (N_L, N_S, N_P, N_J) are the cost weights and number of entries for lane direction changes, number of lane changes, people moving changes, and jobs moving changes, respectively; $e_i = \{0, 1\}$ refers to the current lane direction and $\bar{e_i}$ is the initial lane direction; similarly for number of lanes n_s, 2D distribution of people P, and 2D distribution of jobs J.

A proposed state change is considered to be accepted only if Equation (3.5) is beneath a threshold value. Then $F(X_n, \{Z_1, Z_2, ..., Z_K\}) = C$ is used as the objective function. While this might not yield the quickest convergence it works well in practice. This score function can be defined as the average travel time, CO emissions, or distance to desire traffic pattern.

3.4.5 Traffic Indicators

As an overview, we summarize our interactive traffic editing system, and highlight how the road network can be automatically changed.

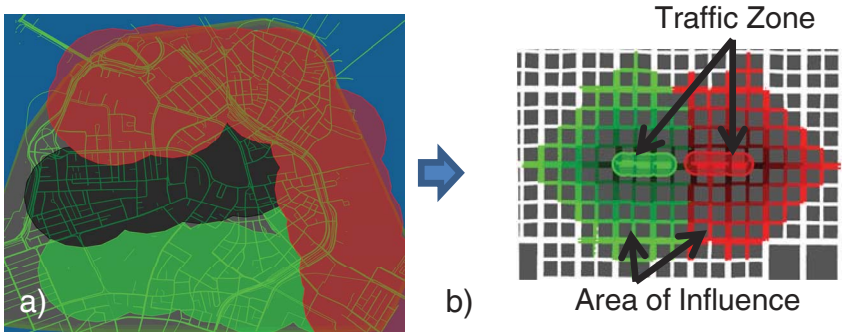

Fig. 3.10. **Traffic Zones.** a) The user can "paint" one or more traffic zones to specify a traffic behavior. b) Each traffic zone has an area of influence that may be altered during the traffic manipulation algorithm.

Traffic Zone Indicator

The designer uses a virtual paint brush to specify a set of K *traffic zones* ($Z = \{Z_1, Z_2, \ldots, Z_K\}$) each with desired constraints or objectives for its contained traffic (Figure 3.10). Each traffic zone is modeled as a union and/or difference of user-drawn circles (or polygons). Further, individual streets can be added/removed from traffic zones. The zone can be specified as constrained (black) or unconstrained. If unconstrained, the paint color indicates whether the traffic is to increase (red) or decrease (green).

Emissions Indicator

Our simulator calculates several indicators which are important for decision-making and policy setting. We measure metrics such as average travel time, distance traveled, and emissions. For example, to report total or per-vehicle CO emission, our system uses the following equation [89]

$$\Lambda = -0.064 + 0.0056 v_m + 0.00026(v_m - 50)^2 \tag{3.7}$$

where Λ is the emission rate (in grams of CO per vehicle per second) and v_m is speed of the vehicle in mph.

3.5 Inverse Design of Weather

Our goal is to create 3D urban models where we can simulate realistic weather behavior over a long time horizon (i.e., days, months, years) and at the same time control the weather. For this, we need to create a weather model with the minimum computational cost that would allow us to i) have realistic and interactive weather simulation, ii) design the high-level behavior of the weather during a day, iii) optimize weather and urban-modeling variables (e.g., for urban planning and design), and iv) simulate weather indefinitely. The weather model requires the ability to handle heterogeneity in landscape and the atmosphere, which forms the atmospheric energetics through buoyancy and vertical motion which contributes to clouds formation from differential heating. Further, the simulation must tightly couple 3D urban procedural modeling with super real-time physically-based weather simulation (e.g., computing a day of weather in significantly less time) (Figure 2.10). In addition, the weather model should provide multiple ways to design weather patterns including using initial conditions specified by users, employing example-based data to generate desired weather conditions (e.g., sunny, overcast, stormy, cloudy), and automatically determining the initial conditions and 3D urban model yielding a desired weather pattern. In other words, our system has to be fast, interactive, realistic, and of value to both the graphics and the urban weather community.

Given a procedurally-generated model, we discover how to alter the model or the initial conditions so as to produce a desired weather. Since weather simulation is a very nonlinear and complex process, it is very hard to predict and/or control weather phenomena. Therefore, we propose an MCMC-based method, in particular, based on the Metropolis-Hasting algorithm [80, 81], to explore the search space to find a solution that exhibits the desired weather behavior. Note that while we use the term

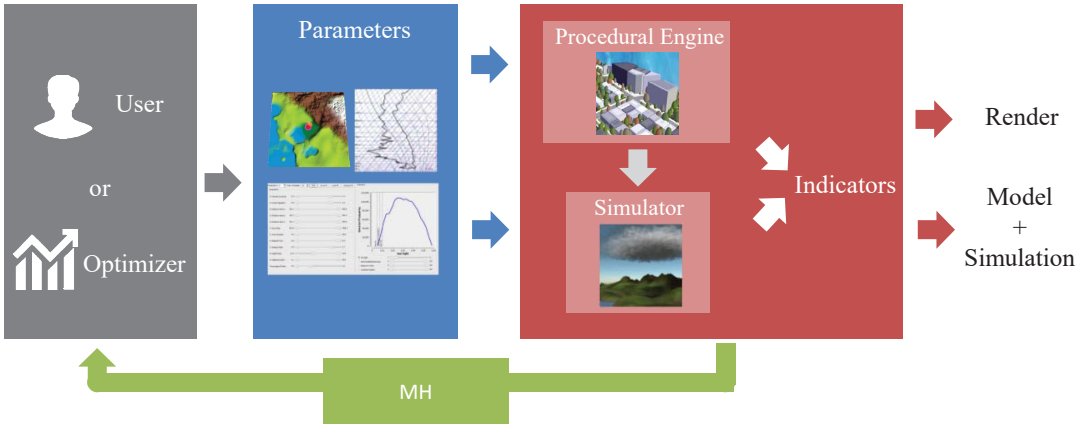

Fig. 3.11. **Weather Pipeline.** Our approach enables a designer to specify a weather behavior and the system will compute what realistic 3D urban model yields that behavior. The user defines a set of input parameters: land use, initial weather conditions, and/or procedural parameters; inputs are used to create the 3D model and simulate weather. Our system iteratively simulates and alters the land use, model, or weather conditions so as to find solutions that meet the desired behavior.

weather the likelihood of desired is more correctly referred to as *climate*. However, we will continue using the term weather as a colloquial usage of the meteorological term.

3.5.1 Weather Seeding Initialization

The user can choose whether to use a single seed consisting of the parameter values of the original model or multiple seeds that consist of physically-based valid parameter sets. In practice, a useful configuration to control the weather without changing the 3D model is to fix $\{\omega_l, \omega_p\}$ and alter the initial weather conditions ω_w. In this case, we create multiple seeds with different ω_w parameter values. Using domain knowledge in weather simulation, we know that significantly varying temperature/clouds/rain can be produced by altering the initial wind (i.e., U) and humidity (i.e., q_v) values at multiple heights. To speed up the convergence and detect non-physically possible

behaviors, we precompute a wide range of physically valid range of values for ω_w (by default 128 in our configuration) and land use distributions (by default 10 different scenarios). These values are then used to automatically select a discrete set of initial seeds close to the objective and, thus, likely to achieve fast convergence. Moreover, we use these precomputed values to evaluate the solution feasibility (See Section 3.5.3).

Given the current state Ω_t, a candidate next state is computed by changing one or more values of the three parameters sets $(\omega_l, \omega_p, \omega_w)$. For parameter values in ω_l and ω_p, the user selects a set of grid cells where the changes can happen (or the whole scenario), the allowable changes of land uses (e.g., change from forest to low-density residential and green), and subset of procedural parameters that can change (e.g., only the window-to-wall ratio and roof albedo can change). To calculate the next state Ω_{t+1}, our system randomly selects a subset of the permissible grid cells and performs a random perturbation to the land use or a procedural parameter in each grid cell. For each parameter value ω and its physically-possible range $\omega \in (\omega_{min}, \omega_{max})$, we perturb the current value with a value sampled from a Gaussian distribution $\mathcal{N}(0, \alpha|\omega_{max} - \omega_{min}|)$ where α is the perturbation change (typically set to a random value $\alpha \in (0 - 10\%)$). For constrained optimization mode, we clamp the final value to enforce the constraints. If the change increases (or decreases) a land use type in a grid cell, the other land use types in the same grid cell are randomly decreased (or increased) by the same amount so that the sum of all land use distributions stays at 100%. For parameter values in ω_w, the user selects the weather variables that can be altered and the plausible physical range (or the systems default physically-based range values are used). Then, using the same sampling scheme, the variables are perturbed.

3.5.2 Weather Acceptance Ratio

We have modified the acceptance ratio and it is computed as

$$a_{min}(\Omega_t \rightarrow \Omega_{t+1}) = \begin{cases} \min\left\{e^{-\beta\frac{E(\Omega_{t+1})}{E(\Omega_t)}-1.0}, 1.0\right\} & \text{if } E(\Omega_{t+1}) \geq \eta \\ \min\left\{e^{-\beta\frac{C(\Omega_{t+1})}{C(\Omega_t)}-1.0}, 1.0\right\} & \text{if } E(\Omega_{t+1}) < \eta \end{cases} \quad (3.8)$$

where β is the energy level that affects the acceptance ratio, $E(*)$ is an error (objective) function, and $C(*)$ is a cost function. If β is small (1.0), then it is more likely that higher error candidate states are accepted. In contrast, when β is larger, it is more likely to accept only improved candidate states.

Our formulation differs from the standard one in two significant aspects:

- Our acceptance ratio is based on relative improvement. For the standard acceptance ratio of Equation 3.1 to work, the error function $E(*)$ should produce similar values during the entire search process, or normalization factors should be computed and used. However, computing normalization factors is very challenging when the error function varies by several orders of magnitude during the optimization iterations. Instead, our approach eliminates the need to find adequate normalization factors by creating an acceptance ratio that only depends on the relative improvement. To accomplish this, our formulation replaces the difference computation of the error functions by a division of them. In Figure 3.12, we show the behavior of the acceptance ratio. The actual values of $E(*)$ are irrelevant only the relative level of improvement (or decline), is important.

- Our modified formulation also serves to implement the error optimization and cost minimization modes (from Section 3.2). In error optimization mode, the function is simply optimized as in standard MCMC. This is accomplished by setting $\eta = -\infty$. In our cost minimization mode, two behaviors will alternate.

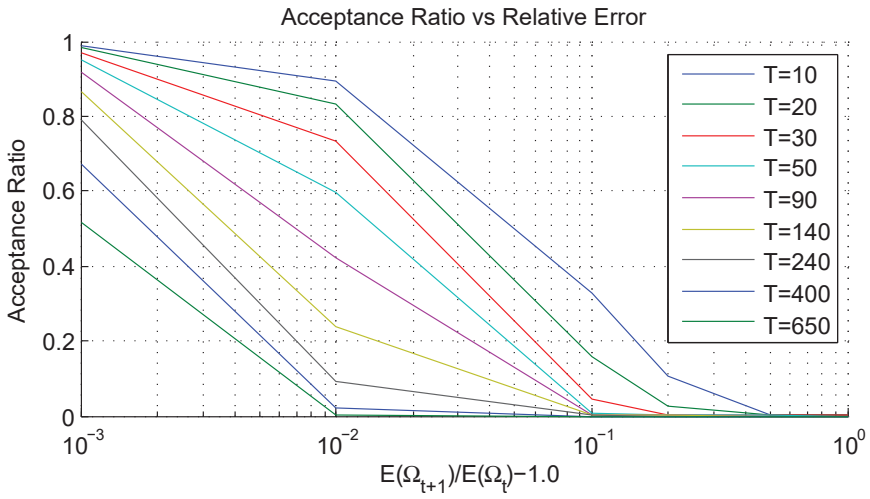

Fig. 3.12. **Acceptance Ratio vs Relative Error.** The figure shows the acceptance ratio for different energy levels. While the temperature decreases, the acceptance ratio decreases as well, making more unlikely that worse states get accepted.

While the $E(*)$ is not satisfied (e.i. $E(*) \geq \eta$), the optimization will minimization $E(*)$. In contrast, when the objective value is achieved (e.i. $E(*) < \eta$), the optimization minimize a second function $C(*)$ (i.e., the cost function). Using this modification, we can optimize a variable but with the minimum number of changes (or cost). Figure 5.16 shows an example – once the temperature objective is reached, the optimization then tries to reduce cost alternating between both behaviors and the search stays close to the $E(*) \approx \eta$. See Section 5.3 for more details.

3.5.3 Towards Global Design

One of the limitations of our current local simulation and inverse approach is that not all possible user-defined weather behaviors are achievable. Variability is constrained by the physical laws we model. The user might define a particular weather behavior that is not feasible locally. This is detected comparing the current scenario

against our precomputed examples as described in Section 3.5.1, if the error is bigger than a threshold, our towards global design approach is used instead.

Our current implementation removes the toroidal boundary conditions and, instead, feeds in external user-defined weather states. These weather states can be easily selected from our precompute states. This ensures their physical feasibility. Then, the user selects several of these states and defines a timeline for each of them. These states are then fed in as changing boundary conditions over time. By forcing the wind to blow inwards across the boundary, we can 'receive' the selected states as an external source collectively yielding the desired weather behavior. Moreover, the user can decide whether the local weather should interact with the external one. This is done by altering the initial soundings: a sounding with a relatively low water vapor mixing ratio and wind speeds similar to the incoming state will provide weather almost identical to the selected external weather. In contrast, using a sounding with a high water vapor mixing ratio and/or different wind speeds will cause interaction between local and external advected and moisture values.

3.6 Example-Driven Road Design

So far in this chapter, we have described how to control the input parameters and modify the 3D model to achieve the desired design. We have shown that our methods work efficiently for models and simulation. However, road networks have an additional component that is hard to capture procedurally, the temporal growth and political and geographic reasons that build up the layout of a city.

We propose a new approach to generating large-scale realistic urban road networks that combines the advantages of example-based and procedural modeling. Creating more realistic road networks benefit the creation of 3D content, in addition to traffic engineering and urban planning. Figure 3.13 shows an example of an intersection generated by our system.

Fig. 3.13. **Road Intersection.** Example of a complex intersection generated by our system. Additional geometry is added for the details such as the sidewalk and urban amenities.

We present an interactive tool that allows untrained users to design roads with complex realistic details and styles. The key inspiration is to recognize that roads are generated based on two processes: i) urbanization of a new area or re-urbanization, and ii) the progressive and more random growth of a city that is expanding. For the former, that usually contains high-level patterns and that has been designed by an urban planner, procedural modeling can encapsulate these complexities and many patterns can be easy created tuning the procedural parameters or using statistical data. For the later, to capture the small details that road networks contain that varies in each part of the city, we propose an example-based growth approach.

Figure 3.14 presents the general overview of our algorithm. The process is as follow:

- The user selects an interesting source network from Open Street Map (Figure 3.14a), in our examples we have used styles extracted from 15 different cities, such as Madrid, San Francisco, Canberra, Tel-Aviv, and London.

- Arterial and local roads are processed independently (Figure 3.14b) to generate a road graph (Figure 3.14c).

- The road graph is divided into *patches* (set of roads with interesting features) and statistical data is computer from the graph (Figure 3.14d). The patches can contain user-specified features (e.g., roundabouts extracted from circular shapes using Hough transform) or interesting intersections (a central node and its connected edges). Figure 3.14e shows the patched found in the given example.

- The user defines the target polygonal area and one or multiple target destination seeds (Figure 3.14f)

- The growth starts from the specified seeds and the process repeats recursively (Figure 3.14g-k).

- The user can select to use the procedural generation. This generation uses the statistical data generated from the source example to tune the procedural parameters. Otherwise, as default, the example-based example will be used. Note that not all patches can be applied at each growth step (the bottom of Figures 3.14f-k shows the potential patches to use). When not possible patch is available, the procedural generation is used.

- Using a similar method, but using as seeds the original position of the example-based, local roads are generated.

- An after-process is executed to clean the resulting network (e.g. remove dead-ends).

Our system has two main ways to interactively created the road network:

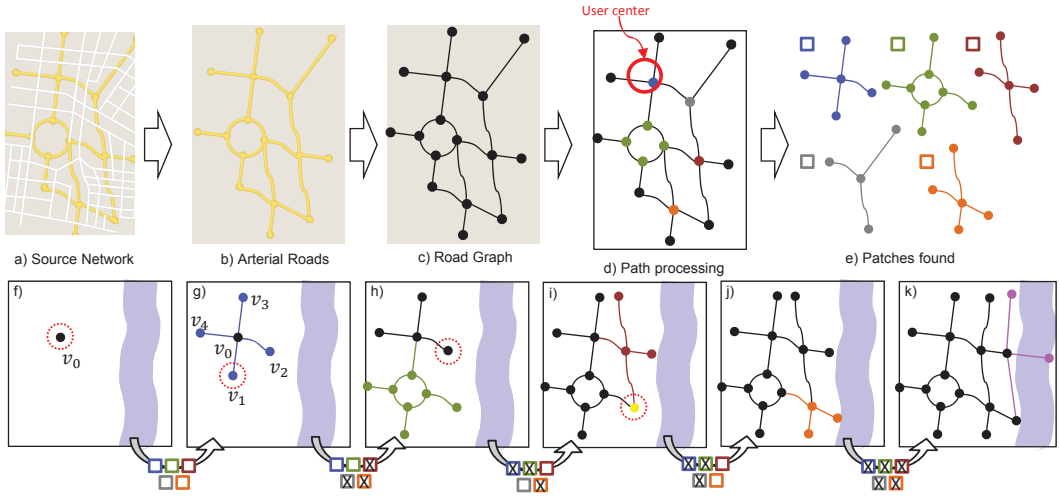

Fig. 3.14. **Example-Based Overview.** a) The user selects an interesting source network. b) The arterial roads are extracted. c) A road graph is generated from the example roads. d-e) Patches are created from intersection and from detected interesting features. f-k) Example of growth, a patch is added between the possible ones.

1. **Sketch Design.** The user wants to generate a realistic 3D model without providing too much detail. The user sketches the terrain, selects several interesting regions as source examples, and specifies the target area. Our system generates a realistic 3D model of a city using the road patterns selected within minutes.

2. **Detailed Design.** The user wants to control the generation of the 3D model or manipulate an already generated model. The user uses our high-level manipulation tools (e.g., warping, blending, and manual drawing) to design the desired road network.

We have used our approach to creating road networks covering up to 200 km^2 and containing over 3,500 km of roads. Figure 3.15 shows a final result of our method. After the road network has been generated, we use our procedural engine to extract the blocks and generate parcels and buildings. Finally, we add urban amenities and vegetation.

Fig. 3.15. **Result of Our Method.** 3D model of our example-based method. After the road network is grown, blocks, parks, parcel, buildings, vegetation, and urban amenities are generated.

3.7 Summary

In this chapter, we have presented inverse procedural modeling and inverse design. We have presented our inverse method based on MCMC to find the necessary input parameter values that creates the desired 3D model, traffic pattern, or weather behavior.

4. GEOMETRIC ASSETS AND VISUALIZATION

In this chapter, we present two methods for automatic reconstruction of buildings densely spanning a city or portion thereof. The demand for such 3D volumetric content has been significantly increased due to the proliferation of urban planning, city navigation, and virtual reality applications (Figure 4.1). Nevertheless, automatic widespread reconstruction of urban areas is still an elusive target. Services, such as Google Earth/Maps, Apple Maps, Bing Maps, and OpenStreetMap have fomented the capture and availability of ubiquitous urban imagery and geographic information system (GIS) style data. Using LIDAR data is one option for city modeling. However, it still has challenges and is not always available. Ground-level imagery provides high resolution but such images are usually scattered and incomplete. Aerial images provide extensive and uniform coverage of large areas, albeit at lower resolution, and are widely available for most cities. Hence, to reconstruct large urban areas we focus on aerial imagery.

4.1 Related Work

In this section, we relate our work to urban modeling approaches in procedural modeling, image-based algorithms, LIDAR-based methods, and volumetric reconstructions including graph-cuts. Image-based algorithms, from computer graphics, computer vision, and photogrammetry, have generated very compelling urban reconstruction results. A recent survey by Musiaski et al. [90] provides an overview of numerous urban reconstruction techniques. Some representative works have created individual facades from images (e.g., Müller et al. [17], Xiao et al. [91], Teboul et al. [92]), individual buildings and statues (e.g., Lafarge et al. [93], Vanegas et al. [16]), interiors (e.g., Furukawa et al. [94]), and point cloud reconstructions (e.g.,

Liao et al. [95]). However, these methods have not produced volumetric building models (e.g., complete texture-mapped building envelopes) of large city areas. Approaches have also been proposed that use large online photo communities to perform reconstructions of popular areas (e.g., Goesele et al. [96], Agarwal et al. [97], Frahm et al. [98]). However, these results are fragmented and cannot necessarily produce all buildings in a given target area. Daniels et al. [99] extract high-quality spline based features but assume a point cloud dense enough to apply RMLS (type of MLS, such as [100]). Chauve et al. [101] determine planar regions and then extend the planes so as to indirectly find edges. They assume points in the same plane belong to the same spatially adjacent cluster.

Numerous methods exploit LIDAR data sources. For example, Nan et al. [102] and Zheng et al. [103] provide interactive tools to improve partial scans of individual building models. Zhou and Neumann [104] provide striking results by extending dual contouring to 2.5D building structures. Poulis et al. [105] present an automatic method to reconstruct 2.5D buildings from aerial images and LIDAR data. They propose a framework using i) 2.5D graph-cuts, ii) automatic and interactive segmentation, and iii) automatic identification and reconstruction of linear roof types. Lafarge and Mallet [106] segment data into ground, buildings, vegetation, and clutter. Then, buildings are formed by fitting points to a collection of template primitives. In general, these methods, and similar ones, rely on the availability of high-resolution point cloud data, sometimes make assumptions of the roof/building geometry, and some do not produce colored/textured models a nave projective texture-mapping using the available aerial images will not necessarily produce good results, as shown in our results subsection. Shen et al. [107] presents an adaptive partitioning of unorganized LIDAR data to find high-level facade structure repetitions. This method can be used to consolidate facades but it is not designed to recover geometry. Toshev et al. [108] detect building structures from city-scale 3D point clouds and construct a hierarchical representation for high-level tasks. Also Golovinskiy et al. [109] present another approach to recognizing objects in 3D urban LIDAR data using specialized

clustering and graph-cut segmentation. However, reconstruction is not the focus of these last three methods. Some methods focus on the registration of aerial images with LIDAR data or with 3D models. For example, Ding et al. [110] describe a new feature called 2DOC based on 2D corners that corresponds to orthogonal structure corners in 3D. Wang et al. [111] improve the registration by using a novel feature called 3CS which uses sets of connected lines. To create a robust registration, they first overestimate the number of line segments and then perform a RANSAC-based refinement. Frueh et al. [112] automatically texture detailed 3D models. They improve the texture discontinuities of each triangle using a classification approach and reduce the graphic card memory footprint using an atlas approach. They present nice results but with clearly visible seams between ground-base and airborne textures. Volumetric reconstruction via space carving, graph-cuts, and related methods have also received significant attention. Methods, such as space carving [113] and image-based visual hulls [114] assume the presence of many images observing the silhouette of the object. Such observations are in general not possible using aerial images of dense urban environments. Another option is using a set of ground-level images to reconstruct the facades of buildings (e.g. Gallup et al. [115] uses a high-resolution video with a priori calibrated street level video and per-pixel depth map as input; Frahm et al. [98] uses a scattered set of images; Grzeszczuk et al. [17] reconstructs building facades from street level images without significant occlusions) but it is impossible to fully sample all facades and all roofs of all buildings in a large urban area. Pollard et al. [116] present a voxel-based volumetric method to detect changes in a 3D scene. Despite presenting some similar inspiration, this approach is designed to detect changes instead of find similarities.

Graph-cuts have been extensively used in computer graphics (e.g., texture synthesis Kwatra et al. [117], Lefebvre et al. [118]). For volumetric reconstructions, graph-cuts are applied to 3D subdivisions of space and combined with stereo processing (e.g., Vogiatzis et al. [119], Sinha and Pollefeys [120], Tran and Davis [121]). Nevertheless, these methods rely on high photo-consistency over the entire build-

ing surface and require an initial building geometry, such as the visual hull. Using aerial images to obtain the visual hull as well as sufficient samples for robust photo-consistency metrics over the entire building surface is challenging for dense urban environments. In our work, we also use graph-cuts, but we define a surface graph-cut that lies on building roofs and walls and on the ground surface. Further, each graph node is positioned and oriented in 3D space but is only connected to its neighboring surface elements. Lempitsky and Ivanov [122] also use graph-cut optimizations, as well as gradient-domain techniques, to address the problem of texture fragmentation on a 3D surface. They assume i) all textures completely see the object, ii) there are no occlusions, iii) all images have the same quality (=importance), and iv) the cameras are perfectly calibrated. These assumptions allow them to simplify their cost function to only use the direction of the corresponding view and the surface normal and to discard any duplicated or overlapping texture segments. Allene et al. [123] alleviate the aforementioned equal image-quality assumption by including optimization terms to measure the effective texture resolution and the color continuity at edges between faces assigned to different (textured) images. Moreover, they use per-pixel blending to minimize the difference due to lighting conditions. In contrast, our approach tackles the problem when occlusions are frequent, camera poses are not contiguous nor have similar angles, cameras are not perfectly calibrated, and the proxy model is not guaranteed to be accurate.

Alternative approaches have been proposed. Gao et al. [124] directly operates on the points and splats/combines results to an output image without obtaining a geometric model. Mathias et al. [125] use structure-from-motion, image-based analysis, and shape grammars. The reconstruction results are promising. However, a grammar is required, which thus lacks automation for large-scale deployment. A related semi-automatic approach is that of Taillandier et al. [126]. However, their method has several requirements which make it not adequate for many urban areas: they only handle square buildings with slanted roofs; they require having an accurate outline of the building and not just the parcel contour or a rough approximation.

In contrast to previous methods, our work focuses on automatically obtaining complete (e.g., closed) building models of urban tall building areas (e.g., downtown, office buildings, financial districts) spanning multiple square kilometers and rely only on aerial imagery and commonly available GIS data for cities around the world. In addition to estimating a building proxy, our method enables the creation of plausible texture-mapped building models using stitched together imagery, even in the presence of imperfect geometric proxy estimates and imperfect camera calibration. Some commercial ventures, such as C3 Technologies (purchased by Apple), pursue similar 3D reconstruction objectives but to our knowledge use manual-intervention and wide-baseline stereo to obtain building models, thus making widespread deployment challenging.

4.2 Volumetric Reconstruction and Surface Graph-Cuts

The demand for 3D city-scale models has been significantly increased due to the proliferation of urban planning, city navigation, and virtual reality applications. We present an approach to automatically reconstructing buildings densely spanning a large urban area. Our method takes as input calibrated aerial images and available GIS meta-data. Our computational pipeline computes a per-building 2.5D volumetric reconstruction by exploiting photo-consistency where it is highly sampled amongst the aerial images. Our building surface graph-cut method overcomes errors of occlusion, geometry, and calibration in order to stitch together aerial images and yield a visually coherent texture-mapped result. Our comparisons show similar quality to the manually modeled buildings of Google Earth, and show improvements over naive texture mapping and space-carving methods. We have tested our algorithms on a 12 square kilometer area of Boston, MA (USA), using 4667 images (i.e., 280GB of raw image data) and producing 1785 buildings.

Fig. 4.1. **Urban Modeling.** A complex urban area (left) is automatically obtained using volumetric reconstruction with surface graph-cuts (middle) computed from aerial imagery and GIS-style parcel/building data (right). Our methodology uses photo-consistency to robustly recreate 2.5D building structures and surface graph-cuts to assemble seamless and coherent textures despite occlusion, geometry, and calibration errors.

4.2.1 3D Urban Reconstruction

There have been several fundamental approaches for producing urban volumetric reconstructions. In contrast to partial (or facade-level) reconstructions (e.g., Müller et al. [17], Xiao et al. [91]), we seek to automatically create texture-mapped building envelopes spanning a large-portion of a city (i.e., akin to the crowd-sourced created models visible in Google Earth) such complete models are suitable 3D content for the aforementioned graphics and visualization applications. Inverse procedural modeling approaches pursue generating parameterized 2D and 3D models from observations (e.g., Šťava et al. [13], Bokeloh et al. [14], Park et al. [74]), but have not been demonstrated for large-scale urban areas due to the inherent complexity and ambiguity in the inversion process. Relevant volumetric reconstruction methodologies from image-based modeling and computer vision can be loosely divided into i) space carving and similar techniques (e.g., Kutulakos and Seitz [113], Matusik et al. [114], Montenegro et al. [127], Lazebnik et al. [128], Shalom et al. [129]) and ii) volumetric graph-cuts

(e.g., Vogiatzis et al. [119]). All of these methodologies exploit, in some form, photo-consistency, visibility constraints, and smoothness assumptions.

However, for our targeted large areas with high building density and thus a high-level of occlusion, we cannot assume a dense, complete, and un-occluded sampling of all building and ground surfaces. These facts about the input data spawn three important challenges. First, although in a typical aerial capture process each building might be at least partially observed in 25-50 images, parts of each facade might only be seen by a few images (and sometimes none at all). This relatively sparse sampling of the building walls hinders photo-consistency measures. Further, the limited visibility and high-level of occlusion also encumbers the silhouette usage and robust foreground/background segmentation for space carving and hampers the determination of the initial geometry (e.g., visual hull) for volumetric graph-cuts. Second, since the captured images of building and ground surfaces may be plagued with the projections of nearby buildings, obtaining occlusion-free projective texture mapping (i.e., texture mapping without neighboring buildings unwillingly appearing on other buildings) would require very accurate geometry. Third, obtaining such very accurate geometry is hindered by camera calibration error and by the grazing angle observations of the building facades. Nave projective and view-dependent texture mapping would produce strong visual discontinuities or would compensate for the inaccuracies by using significant blending/blurring. Our solution circumvents the aforementioned challenges by exploiting the following inspirations.

- Buildings are, by and large, individual 2.5D structures; thus we assume each successive floor up the building is equal to or contained within the contour of the previous floor.

- Since aerial images mostly sample the roof structures of a building, we exploit photo-consistency only for determining the roof structure; for the building walls, we exploit the 2.5D assumption and stitch together the visual observations using a surface graph-cut based technique (a surface graph-cut is a 2D

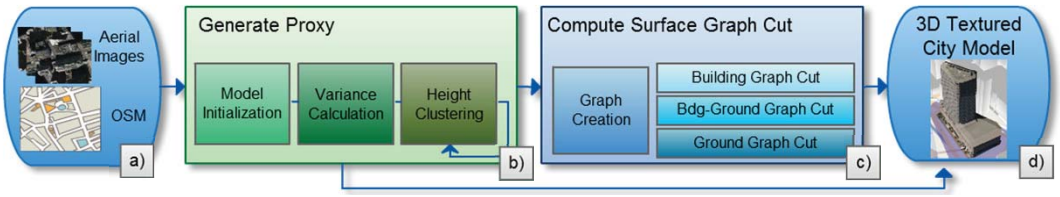

Fig. 4.2. **System Pipeline.** Our system uses (a) aerial images and GIS-like input data to (b) compute a geometric proxy, (c) generate surface graph-cuts, and (d) assemble textured 3D building models of large urban areas.

manifold in 3D space that has been stitched together using a solution to the minimum-cost graph-cut problem); our surface graph-cut assembles a seamless and visually-coherent texture-mapping of the buildings and ground surfaces despite an imperfect building proxy, projected occlusions, and camera calibration errors.

- To solve the chicken-and-egg dilemma of needing to know the geometry to solve for visibility (and needing to know visibility to solve for geometry), we exploit the assumption of having approximate GIS data (e.g., building outlines) in order to formulate simple building shape estimates which we enhance. Our approach builds upon voxel occupancy and graph-cuts (e.g., Kwatra et al. [117]) to automatically and robustly yield large-scale 3D urban reconstructions. Our largest example includes 1785 reconstructed and texture-mapped buildings spanning more than 12 square kilometers. Our system pipeline (Figure 4.2) takes as input a set of pre-calibrated high-resolution aerial images captured from a multi-camera cluster flying over a city (courtesy of C3Technologies), approximate building outlines extracted from a GIS provider (i.e., OpenStreetMap (OSM)) and rough initial building heights per city zone.

A coarse initial building geometry is subdivided into voxels which are then refined. Improved building outlines, heights, and roof structures are obtained by using a

photo-consistency and clustering algorithm. Then, we use surface graph-cuts to add the remaining visual details to the building walls and the ground. The roof structure is sampled by many images. Thus, texture mapping the roof voxels to display additional visual details can be straightforwardly done by selecting the most head-on observations. However, the building walls are sparsely sampled. Hence, in order to create a complete, coherent, and occlusion-free colored appearance, we texture-map wall voxels using the aerial images for which a satisfactory graph-cut with the roof and with the adjacent building walls is produced. Further, we solve two other surface graph-cut problems in order to provide a smooth visual transition between the building walls and the ground surface as well to produce a top-down high-resolution ground surface image that is free of unwanted projections of building geometry and shadows. Altogether, our method exploits photo-consistency only where it is highly sampled (thus less susceptible to outliers and noise) and uses a graph-cut based algorithm to stitch together a visually plausible result for the rest of the building surfaces and the ground surface. Our examples are from a large metropolitan area (i.e., Boston, MA in the USA) using a dataset of 4667 aerial images and conservative initial building outlines and height estimates (e.g., often overestimates of 50%). Our comparisons show that our results are significantly better than texturing a space-carving/visual-hull result and similar quality to crowd-sourced manual modeling efforts.

4.2.2 Voxels and 3D Graph-Cut

We identify two main tasks to reconstruct a city: i) find the building geometry (Figure 4.2b), and ii) texture the models (Figure 4.2c). For the first task, we could discretize the space of the whole city and try to find the geometry at the same time, but this approach would not scale since the number of voxels is linear with the size of the city (e.g., in our case it would be $O(10^8)$ voxels). Given the individual nature of each building (i.e. a building can be seen as an independent model surrounded by streets), we simplify the problem by processing each building individ-

ually. For each building, we first initialize the building with a set of voxels using the GIS data (subsection 4.2.3) and find the photo consistency-between aerial images (subsection 4.2.3). Then, we find 2.5D building geometry (subsection 4.2.3). For the second task, we could use a standard view-dependent texture mapping, but this would assemble imagery by blending together fragments from many different images. Such a method does not exploit the internal consistency of each captured image and might create seams along image transitions. In contrast, we use graph-cuts to stitch together imagery from as few images as possible so as to exploit internal consistency as well as produce seamless texture mapping. Given that the complexity of graph-cuts (solved using the min-cut algorithm) is $O(VE^2)$, it would not scale to city level (in our case it would be $O(10^19)$). Therefore, we also process each building individually. However, this does not completely solve the problem since the ground should be also textured, which in turn necessitates a smooth transition between building and ground surfaces. To overcome the problems of this task, we use graph-cut for three different purposes: i) texturing building surface (subsection 4.2.4), ii) improving the transition building-ground surfaces (subsection 4.2.4), and iii) finding an optimized ground surface (subsection 4.2.4).

4.2.3 Volumetric Building Proxy

We first describe our algorithm for computing a per-building volumetric proxy. Our method initializes each building model as a grid of voxels, calculates a weighted photo-consistency measure per voxel, and clusters the voxels of minimum variance. The output is a 3D un-textured proxy model.

Appearance Editing

Each building is initialized as a 3D array of voxels (Figure 4.3a). The voxels are obtained by subdividing a vertical extrusion of a coarse estimate of the 2D building

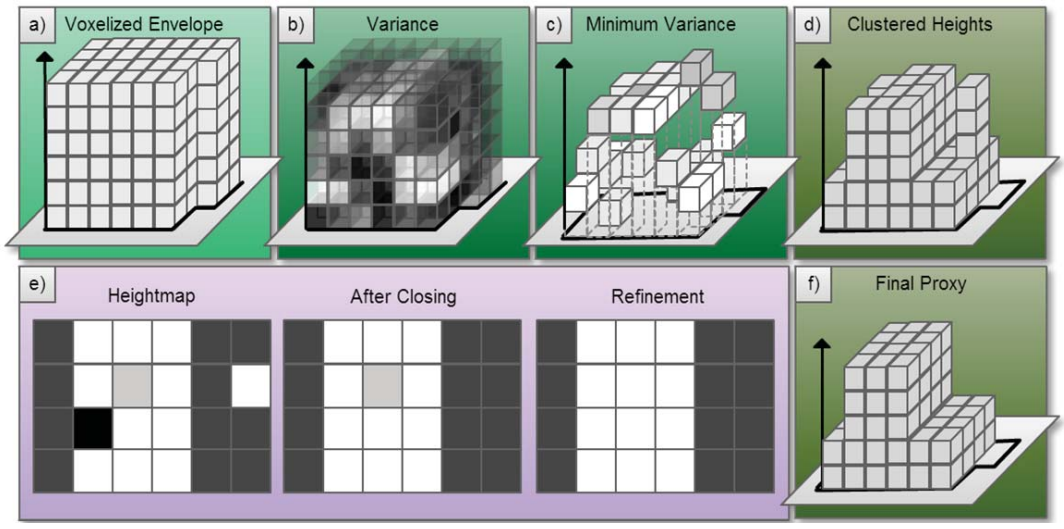

Fig. 4.3. **Building Volumes.** We show the steps of our volumetric building reconstruction. a) An initial model is divided into voxels. b) The per-voxel variance of our weighted photo-consistency measure is computed. c) The most consistent voxel per column is chosen, potentially reducing building height. d) The voxels are clustered by height, e) placed in a height-map, and filtered. f) The final proxy model is obtained.

footprint. Given a building of size $[b_x, b_y, b_z]$ and a voxel size r, we label each voxel v_i for $i \in [1, N]$ and $N = (b_x/r) * (b_y/r) * (b_z/r)$.

For notational brevity, we also assume v_i refers to the 3D position of the middle of voxel i. The upper bound for N is when a voxel of size r_0 corresponds roughly to one projected pixel. In practice, we choose values of $r > r_0$ in order to reduce the per-building computation time which is important when processing city-scale environments.

Building footprints and building heights, or estimates thereof, are frequently present in a city GISs and some navigation service databases. With regards to building footprints, one option is to use the shape of the enclosing parcel which is roughly of the same shape for dense urban areas. In our case, we make use of the increasingly popular open data repository OpenStreetMap.org. It contains top-down street,

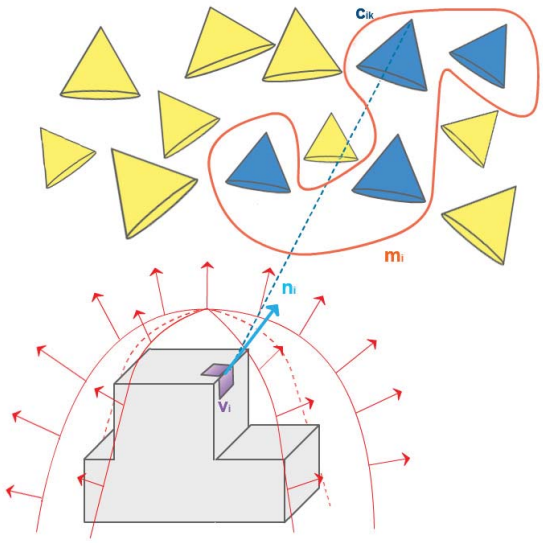

Fig. 4.4. **Variance Calculation.** Using the initial voxel normals n_i for a voxel v_i, we determine the variance of our weighted photo-consistency measure of the subset of cameras, such as c_{ik}, that best see the voxel.

parcel, and approximate building outlines for a very large number of cities worldwide (as seen in Figure 4.2a and in our video). We extract building outline estimates from images such as this using image processing; in particular, we detect a loop of edges per parcel and form a closed polyline. For building heights, if not available in the GIS, we make zonal estimates (e.g., residential zone apartments are given a constant height, high-rise zones are given a higher constant height); however, the building height should be conservative (e.g., we frequently overestimate height by 50%). Photo-consistency will enable finding the actual roof heights and building outlines.

We must also establish an initial surface normal per voxel. After inspecting many buildings, we found that a good prior is to represent a building as a half ellipsoid (Figure 4.4, bottom). At this stage in the pipeline, the voxel normal is solely used to determine the subset of the aerial images that potentially see the voxel. This

approximation does not directly affect the resulting building geometry but rather helps select which images are used in later stages. Because it is not known yet which voxels will be on the building surface, normals are computed for all voxels of the initial model (i.e., interior and exterior voxels). Given a building, we first fit the upper-half of an ellipsoid to the building by computing values for the ellipsoid radius and ellipsoid coefficients a, b and c. Then, given voxel v_i, we compute the voxels initial normal as

$$n_i = \left(\frac{2v_{i_x}}{a^2}, \frac{2v_{i_y}}{b^2}, \frac{2v_{i_z}}{c^2} \right) \tag{4.1}$$

Given voxel positions and normals, we obtain the color c_ik for voxel i observed by camera k. To support different voxels sizes (both when voxel-to-camera distance varies amongst the aerial images and when purposefully working with larger voxels to increase reconstruction performance), we project the voxel onto camera image k, estimate the size s_{ik} of voxel i on camera image k and grab a Gaussian weighted footprint of pixels as

$$c_ik = \sum_{t \in [(-s_{ik}, -s_{ik}), (s_{ik}, s_{ik})]} proj_k(v_i) + t) e^{-||t - proj_k(v_i)||^2 / 2\sigma_c} \tag{4.2}$$

where $proj_k()$ returns the projection of its argument onto camera image k and σ_c is the standard deviation of the Gaussian. Given s_{ik}, σ_c is obtained by the known estimate $0.3(s_{ik}/2 - 1) + 0.8$

Variance Calculation

Starting with the initial model of a building, we search for a subset of voxels that are photo-consistent amongst the aerial images observing the building. We assume strong photo-consistency for a voxel implies it is on the actual building surface. As the measure of photo-consistency, we use the weighted variance of the color of a voxels projection on different aerial images (Figure 4.3b).

In preliminary experiments, we investigated several measures for evaluating whether a voxel is on the building surface. We attempted using color-based segmentation of

aerial images and/or the weighted sum of the measures of photo-consistency, local surface planarity, and local supportability (i.e., probability that a voxel is needed because another higher-up voxel will be selected). However, we observed that the various variants of this combined metric were not robust to noise/errors and in practice over-constrained voxel selection. This is primarily due to the relatively sparse (and often at grazing angles) sampling of building walls. As mentioned in the introduction, we did, however, observe many visual samples and significant photo-consistency amongst voxels on the building roof surface which led us to rely mostly on them.

Our method transforms all aerial images to HSL color space and uses the H and S channels. We use only the H and S channels in order to ignore the effect of changing daylight illumination and, to a lesser degree, the effect of shadows. Further, we explicitly weigh variance by the inverted building height of a voxel. Hence, given an approximately tied variance, the vertically higher voxel is chosen. Numerically, our voxel variance measure is computed as

$$m_i = \frac{v_{i_z} + b_z/2}{b_z} \left(\sum_{k=1}^{s_i} c_{ik}^2 - \left(\sum_{k=1}^{s_i} c_{ik}^2 \right)^2 / s_i \right) \tag{4.3}$$

where it is assumed the building is centered at the origin, the first term computes the ratio of the voxels vertical height (assumed to be along the z-axis) to the buildings z size, and s_i is the number of camera images that have a line of sight to voxel i. In order to improve the variance calculation, we use the initial footprints to account for the potential occlusion of neighboring buildings. Specifically we create a mask m_k by rendering the building from the point of view of a camera k pointing towards the building; the building is rendered in white and the background in black. When computing color c_{ik}, we check in the corresponding mask whether the image pixel is white (unoccluded) and should be used, or black (occluded) and should be discarded.

Height Clustering

In aerial images, roofs are expected to be viewed by more cameras than facades (i.e., more photo-consistent). Thus, we find the height of each column by searching for the columns voxel with the lowest variance m_i. We use this information to assemble the final building proxy (Figure 4.3c). If we observe the building from a side, the voxels should collectively exhibit a compact distribution around the different heights of the buildings. Hence, we can use 1D k-means clustering to find those different building roof heights (e.g., $k = 1, 2, 3,)$. Since the optimum value for k is not known a priori, we estimate it using a heuristic that works well in practice. Starting at $k = 1$, we increase k until we find that at $k + 1$ the clustering error reduces by no more than c_e percent. In preliminary experiments, such a clustering algorithm worked well, yielding buildings with 1 to 5 different roof heights. For our results, we set $c_e = 0.3$. After clustering, our method selects the voxel per column whose height is closest to the corresponding clusters mean height (Figure 4.3d).

After clustering, we place all voxel heights into a height map image. Starting with the uppermost cluster, our algorithm performs a per-cluster morphological close operation [130] (i.e., dilate and then erode) in order to remove small islands of the current cluster type and to fill-in small gaps. We also perform an in-filling refinement step to remove any remaining single-voxel holes with no height/cluster assignment (i.e., we find the most popular cluster assignment of the neighboring voxels and assign that value to the missing voxel). Voxels physically below the filtered minimum variance voxels are marked. Then, all exterior surface voxels are selected as being part of the building envelope (Figure 4.3f). Although voxels are small, we reduce jaggedness by adding quadrilaterals to connect corners of adjacent voxels on an off-axis (i.e., diagonal) building surface. It is worth noting that the final proxys outline will not necessarily match that of the initial conservative building estimate. Finally, the voxel normal is recomputed for each exterior surface voxel by summing up the vectors from the voxel center to each existing neighboring voxel, reversing the normal direction,

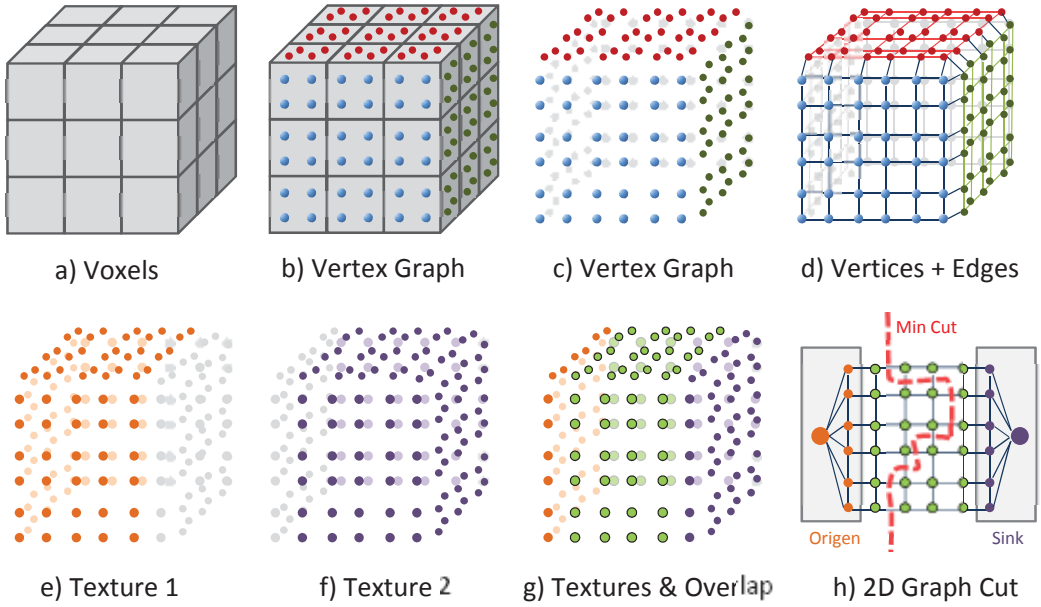

Fig. 4.5. **Surface Graph-Cut.** a) Voxels, b) voxels showing graph vertices, c) vertices, d) vertices with edges, e) vertices seen by an image 1, f) vertices seen by an image 2, g) vertices that see image 1 and image 2 are in green and are where graph-cut will be applied, and h) a 2D graph-cut.

and normalizing the vector. Afterward, the normals of all voxels are averaged using a Gaussian weighting of the nearby voxel normals. Succinctly, this is computed as:

$$n_i = \sum_j \gamma_{ij} norm \left(- \sum_k \delta_{jk} (p_j - p_k) \right) e^{-||p_i - p_j||^2 / 2\sigma_n} \tag{4.4}$$

where σ_m is the standard deviation of the desired smoothing neighborhood, $\delta_{jk} = 1$ if v_j and v_k are adjacent, $norm()$ returns the vector normalized version of its argument, and $\gamma_{ij} = 1$ if v_i and v_j are within $2\sigma_n$ voxels of each other (e.g., 95% of the neighbors that affect normal averaging are considered).

4.2.4 Surface Graph-Cuts

In this subsection, we define surface graph-cuts as well as describe our multiple uses of them. Graph-cuts can be used to solve problems such as image stitching and image segmentation. To solve the stitching problem, a 2D graph is created where each vertex represents a pixel and edges connect adjacent pixels with a calculated weight (e.g., the color difference). The best stitching possible will be the one that minimizes the visible transitions (i.e., the minimum cut through overlapping areas – Figure 4.5h). We extend this idea to not just define a flat image but instead pixels on the 3D surface formed by the visible faces of the building, the interface between building and ground surfaces, and the ground surface. Conceptually, this can be viewed as covering the building with pieces of cloth. Each image is a piece of cloth that partially covers the building. We try to cover the whole building with the least noticeable transitions. The challenge is in choosing cloths and in how to cut them.

Definition

A surface graph contains the visible faces of a volumetric building proxy (Figure 4.5a) and/or of the surrounding ground. Since, for reconstruction performance reasons, we typically chose a voxel size that projects to larger than one pixel, each exterior (i.e., visible from the outside) voxel face is subdivided into SxS subfaces (in Figure 4.5b, $S = 2$) to ensure the final model is textured at near the original image resolution despite using lower-resolution voxels. In each visible face of each voxel, we place a SxS array of vertices (Figure 4.5c). Each vertex v_a is then connected to its neighbor v_b by an edge e_{ab} (Figure 4.5d) to form the 3D graph where a graph-cut will be applied. Thus, the surface graph $G = \{V, E\}$ is composed of vertices $V = \{v_a\}$ for $a \in S^2 N_S$ (where N_S are the faces of the voxels that are in the surface) and edges $= e_{ab} : v_a$ and v_b are adjacent.

Within each graph vertex v_a, our system stores

- q_a: 3D position of the graph vertex,
- n_a: surface normal in the vicinity of the subface,
- k_a: camera image id to use for this voxel,
- c_a: current color of the graph vertex, and
- p_a: potential color of the graph vertex.

A graph-cut defines a smooth visual transition between two adjacent surface *patches*. Each of the two patches is a subset of the surface graph that has the same camera image id (Figure 4.5e and Figure 4.5f). These two patches overlap in a region (Figure 4.5g). The graph-cut process will find the trajectory, in this overlapping area, along which the sum of the color differences between the corresponding pixels of the two source camera images is minimal (Figure 4.5h).

To avoid re-creating the graph for each texture, we create the graph just once and update the weights, origin, and sink before calling the min-cut procedure. To efficiently compute our large min-cuts (e.g., $O(10^6)$ vertices)), we use the augmenting path algorithm of Boykov and Kolmogorov [131] which in practice is significantly faster than other methods. To calculate the cost, we perform color differences in perceptually linear LAB color space in order to improve the perceived transition from one texture to another and not just reduce the numerical color difference (i.e., reducing the Euclidian distance in this color space maps to a perceptual improvement). We define the matching quality cost C between two adjacent vertices s and t that belong to two different patches P_1 and P_2 as

$$C(s,t,P_1,P2) = ||P_1(s) - P_2(s)|| + ||P_1(t) - P_2(t)|| \tag{4.5}$$

where $P_i(*)$ evaluates to a LAB color.

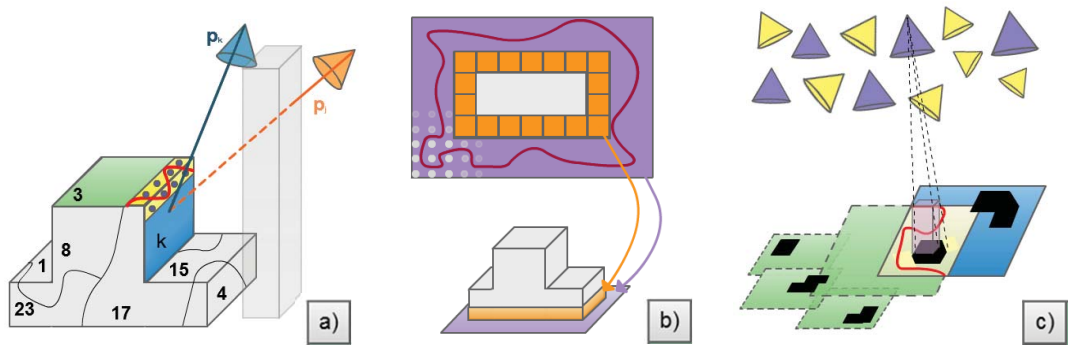

Fig. 4.6. **Applications of Surface Graph-Cuts.** a) We show several patches over a building surface. Each patch is obtained by grouping adjacent subfaces best observed by the same camera while taking visibility into account (e.g., patch k is best observed by camera p_k because p_j is occluded. In this step, patch 3 and k are joined as in Figure 4.5h. b) Another surface graph-cut is defined and computed at the boundary of the building with the ground surface. c) Finally, a ground surface graph-cut is also performed so as to obtain a seamless and free-of-projected-buildings ground texture.

Building Surface

We solve the graph-cut problem for the building surfaces resulting in the best seamlessly stitched texture-map over the building surface (Figure 4.6a). First, we compute which cameras are visible from each graph vertex and choose the visible camera that best samples the vertexs surface fragment. Since we have a very large number of vertices (e.g., over 100,000 per building), we use the graphics card to quickly determine which are visible from each of a nearby set of camera viewpoints. For efficiency, we render each voxel as a color-coded quadrilateral. From all the cameras k, at position g_k, that see a particular voxel and all its subfaces/vertices, the camera k_a for which $(g_k - q_a)n_a$ is maximal is chosen; e.g., $k_a = k$.

Second, spatially-adjacent vertices with the same camera image id are grouped into patches and sorted by size. To assist with reducing the effect of image-to-image illumination changes and calibration errors, we wish to have as few textures and

graph-cuts as possible. Thus, we group same-image-id vertices. We also sort them by area from largest to smallest because the largest group is mostly likely to reference the best aerial image. Empirically, buildings are stitched together from 3-10 different aerial images.

Third, our method assembles the surface graph-cut starting with the largest patch. Given the current processed vertices, the system iteratively searches for the largest adjacent patch. An overlapping frontier is defined within the two patches. Although we could use the entire overlapping area to find the graph-cut, we limit the overlapping area so as to keep most of the current processed vertices intact. Before calling min-cut, we update the weight of each edge: the vertices that have been processed are connected to the origin of the min-cut and their weights are set to infinity (i.e., to not be cut), the edges of the vertices within the transition region are updated with the cost C (between the current color and the new potential one), and vertices that belong to the potential texture but do not overlap, are connected to the sink and their weights set to infinity (i.e., also to not be cut) – as in Figure 4.5h. Our system uses min-cut to search for the cut that minimizes the visual image transition from one patch to another one. This step is repeated iteratively until all vertices have an assigned camera image id. We choose this greedy approach over other global optimizations because i) we do not try to minimize the transition between vertices but between patches, ii) a global (patch) optimization would require an exhaustive/stochastic exploration, iii) it is guaranteed to converge, and iv) it fits the requirement to keep the number of patches as small as possible to minimize the change-of-illumination issue.

Building-Ground Surface

Next, we solve the graph-cut problem for connecting the building surfaces to the ground surface (Figure 4.6b). We extend the building surface by generating a ring of voxels around the ground-level height of the building such that the top most face of each extended voxel coincides with the existing ground surface (i.e., the voxel center is

essentially slightly below ground). Even though the width of the ring can be altered, we use a constant value for all our examples. For each of the newly created voxels, we assign a camera image id to it. This is done by finding the closest voxel on the building surface and copying the camera image id to each of the S^2 graph vertices of the new voxel. To build the local ground surface, we use the same extended building surface vertices but calculate their color using the improved ground surface image (see next subsection). We define a single building-ground graph (per building) with a source node inside the footprint and the exterior ring of voxels connected to a sink node. The graph-cut computes the smoothest transition from building wall textures to ground surface images. The cut is constrained to lie on the ground surface so as to prevent the building textures from changing.

Ground Surface

To produce an improved top-down view ground surface image, we stitch together the most downward facing aerial images (Figure 4.6c). In a manner similar to 2D texture and image synthesis, the aerial images are pieced together sequentially in random order – the order does not matter as long as the ground surface is fully sampled. Since the graph of one ground image is very large (e.g., over a million graph vertices), only a subset of the overlap region between the currently stitched image and a new image is used. A graph-cut is calculated within the overlap region and stored.

To avoid the appearance of building surfaces projected onto the ground outside of the building footprint, we make use of the building proxies. We render each building proxy in black from each images center of projection and onto the aerial image. Then, we explicitly prevent the graph-cut from using, or going through, the building by placing very large cost penalties when choosing to transition to a building pixel. Although it is not guaranteed that all ground surface points are observed, unobstructed, from

an aerial viewpoint, in practice it is possible. The final result is one single coherent, occlusion-free top-down image of the city.

4.3 Automatic Modeling of Planar-Hinged Buildings

We present a framework to automatically model and reconstruct buildings in a dense urban area. Our method is robust to noise and recovers planar features and sharp edges, producing a water-tight triangulation suitable for texture mapping and interactive rendering. Building and architectural priors, such as the Manhattan world and Atlanta world assumptions, have been used to improve the quality of reconstructions. We extend the framework to include buildings consisting of arbitrary planar faces interconnected by hinges. Given millions of initial 3D points and normals (i.e., via an image-based reconstruction), we estimate the location and properties of the building model hinges and planar segments. Then, starting with a closed Poisson triangulation, we use an energy-based metric to iteratively refine the initial model so as to attempt to recover the planar-hinged model and maintain building details where possible. Our results include automatically reconstructing a variety of buildings spanning a large and dense urban area, comparisons, and analysis of our method. The end product is an automatic method to produce watertight models that are very suitable for 3D city modeling and computer graphics applications.

4.3.1 3D Urban Reconstruction

3D city modeling has become extremely popular due to the increased number of computer graphics applications in the entertainment industry, urban planning, digital mapping, and virtual environments. However, the automatic modeling of large dense cities, including the robust recovery of sharp edges and planar features, and the creation of water-tight geometric models suitable for texture mapping and interactive applications remains elusive. Previous efforts have addressed our goal using one or more different input sources (e.g., LIDAR or image data), and from

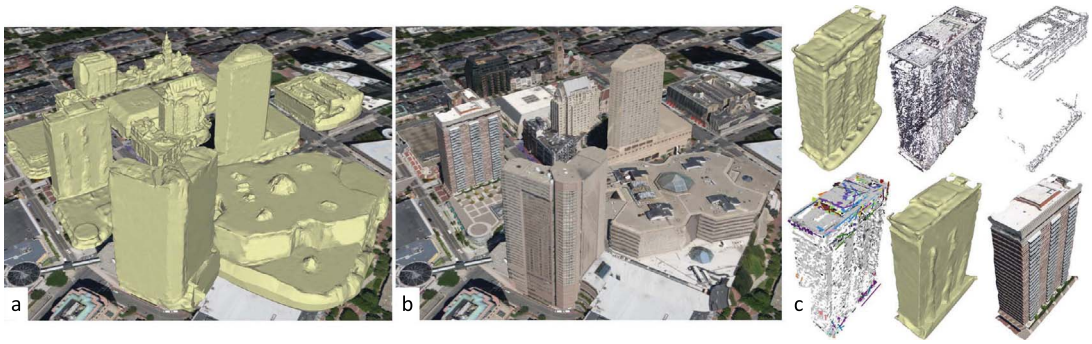

Fig. 4.7. **Reconstructed Buildings.** We automatically reconstruct buildings from images by assuming a building consists of arbitrary planar segments inter-connected by linear (i.e., straight) hinges at any angle. a) Final re-constructed model, b) with projected texture mapping, and c) processing pipeline: initial point cloud, initial triangulation, Canny edge points, visualization of plane/hinge constraints, final model, with projective texture mapping.

ground-level viewpoints, airborne viewpoints, or combinations thereof. We identify three key challenges: i) sampling completeness - obtaining samples from all surfaces is a daunting task typically addressed by either coalescing information from multiple viewpoints or filling-in holes; for large-scale city modeling obtaining a full sampling of all surfaces using multiple viewpoints is impractical; ii) surface triangulation - in the presence of missing samples, generating a closed-triangulation can be hard for traditional triangulation methods which assume clean and near-uniform sampling (e.g., [132]); and iii) noise - in general recovering sharp edges and other surface features (e.g., planarity, circularity) is hard in the presence of significant noise.

Our approach addresses the sampling completeness, surface triangulation, and noise challenges by defining a class of buildings which supports sharp edges and planar segments and using a new framework to improve automatic building surface reconstruction. Coughlan and Yuille [133] defined Manhattan World (MW) buildings as a restricted subset of buildings consisting of exterior facade segments belonging to one of three orthogonal planes. Schindler and Dellaert [134] extended the Man-

hattan World assumption to Atlanta World (AW) which includes multiple groups of orthogonal vanishing directions. We further extend the assumption to arbitrary planar-hinged building models. A building's surface consists of arbitrary planar segments interconnected by linear (i.e., straight) hinges at any angle. This framework affords a more general class of buildings than MW or AW.

We demonstrate several automatically reconstructed buildings within $0.5\,km^2$ in Boston (USA) using 135 aerial images.

4.3.2 Planar-Hinge Modeling and Reconstruction

Our approach uses high-resolution aerial images captured from a multi-camera cluster flying over a city (courtesy of C3Technologies), approximate building outlines extracted automatically from OpenStreetMap (GIS data), and automatically produces a 3D triangulated model.

Initial Model

First, we obtain a *dense 3D point cloud* and an initial *model triangulation* and *model vertices*. The aerial images observing a building are given to Bundler [135] to obtain a sparse point cloud. Then, we use CMVS [136] in combination with PMVS [137] to generate a dense 3D point cloud. The dense 3D point cloud is used to generate an initial model triangulation using Poisson surface reconstruction [138]. Poisson surface reconstruction is able to generate a watertight closed mesh even in the presence of significant missing surface samples. However, the reconstruction generates a new set of approximating vertices, which we call the *model vertices*.

Plane Construction

Next, we find the most probable planar segments in the building's dense 3D point cloud. Initially, we smooth the normals estimated by CMVS using a normalized

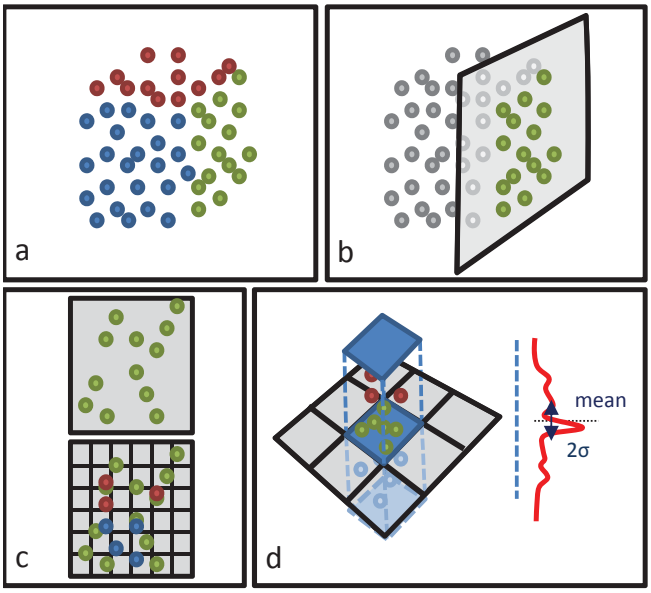

Fig. 4.8. **Planes reconstruction.** a) Segment point cloud b) RANSAC plane fitting c) Projection of points into plane to define bounding box and create grid d) For each cell create a prism to define the distribution of the points inside the likelihood the cell plane exits.

(based on the confidence values calculated by CMVS) bilateral-filter. Then, we find planar segments in the point cloud by growing regions based on point color and normal similarity and use random sample consensus (RANSAC) to determine the plane per region. Our method uses as region starting seeds the most densely-sampled 3D point cloud regions and successively adds points to regions based on normal similarity and distance measurements. After region growing, another pass regroups the regions that form the same plane but are not contiguous (Fig. 2a). Then, we re-run RANSAC to find the most probable plane per group (Fig. 2b). With just a few hundred iterations, the algorithm converges quickly and with a relatively small error (i.e., 5 cm. for up to 200m. high buildings). To preserve small geometric details, we partition the points in each of the aforementioned planar groups into a set of grid cells (Fig. 2c) and determine which cells contain points most likely belonging to the plane (Fig. 2d).

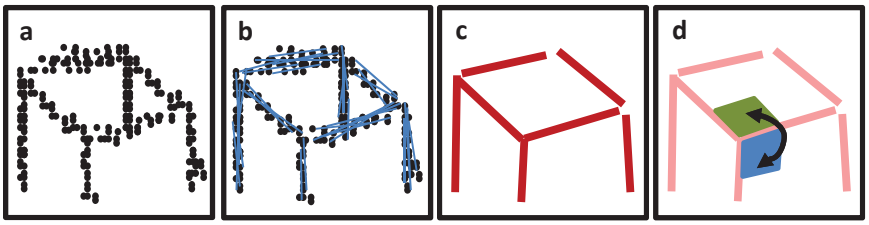

Fig. 4.9. **Hinge reconstruction.** a) Canny point cloud b) Find lines c) Fitted lines d) Using the lines find the hinges: edge and planes

Hinge Construction

Using the input images with a Canny edge detector applied, we run CMVS and PMVS again to generate the building's edge 3D point cloud (i.e., 3D points reconstructed only on the edges of the building) (Fig. 3a). A hinge implies points forming a 3D line segment from location A to location B. There should be points approximately uniformly distributed within a cylinder from A to B and with a small radius r. To find candidate cylinders, we set each point in the cloud as a potential location A and search for a point B such that there are points contained within the cylinder from A to B. When a candidate cylinder is found, it is extended along its axis in both directions until there are no more points in the extended directions (Fig. 3b). Nearby cylinders with similar central axis directions are grouped. Then, we use RANSAC to find the most probable 3D line segment within each cylinder group (Fig. 3c). To find the planes adjacent to each 3D line segment, we partition the line segment and for each partition analyze the corresponding points in the building's dense 3D point cloud within the line segment's cylinder. We fit a single plane to all points in the cylinder segment. Then, we divide the points into two subgroups using the plane perpendicular to the fitted plane. We now fit a plane to each subgroup. If the two fits each have a small error (e.g., 5 cm or less), we have found the two local building surface planes. The result is a set of hinges over the building surface, each being a 3D line segment and two adjacent planar regions (Fig. 3d).

4.3.3 Model Reconstruction

In this process, we modify the model so as to better satisfy the plane and hinge constraints calculated with the 3D point clouds. Each hinge is divided into segments and each segment attracts model vertices within an action radius of its center line segment and to each of the hinge planes. In addition, each plane constraint also pulls model vertices towards the corresponding plane. In summary, the new position p_{i+1} of a point is calculated from p_i as:

$$p_{i+1} = p_i + \sum_{s \in H_l} \frac{(s - p_i)\, k}{\|s - p_i\|^2} + \sum_{f \in H_p} \frac{(f - p_i)\, k \cdot \delta}{\|f - p_i\|^2} + \sum_{c \in H_p} \frac{(c - p_i)\, k \cdot \gamma}{\|c - p_i\|^2}$$

where s is the center of each hinge line segment, k is a spring constant, f is the center of each hinge plane, δ=1 when the point lies within the action radius of the hinge plane and 0 otherwise, c is the center of each plane cell, and γ is the dot product of the grid cell normal with the direction of the point to the center of the plane.

4.4 Summary

In this chapter, we have presented two methods to recover the 3D geometry of a 3D urban model. The results can be used for visualization, urban planning, and educational purposes.

5. RESULTS AND ANALYSIS

In this chapter, we present the results of the 3D urban design (Section 5.1), traffic simulation (Section 5.2), weather simulation (Section 5.3), and geometric assets (Section 5.4).

5.1 3D Urban Procedural Modeling

We use our forward and inverse framework to create, design, and edit a variety of city-scale models. All rendering and editing are interactive, and the results appear while editing. All example sessions were completed in under five minutes and most took less than three minutes.

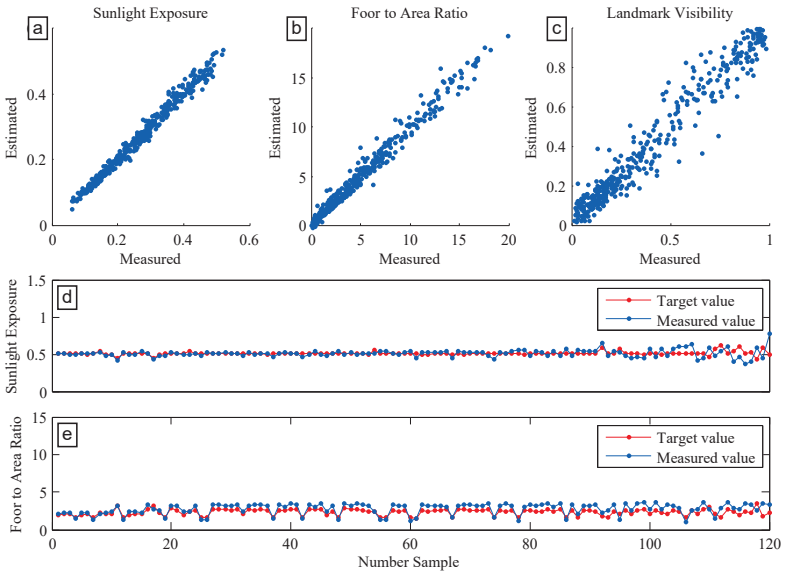

Fig. 5.1. **ANN Error.** a-c) Comparison of measured vs. estimated by the ANN for three different indicators. d-e) Comparison of top 120 target and measured indicator values.

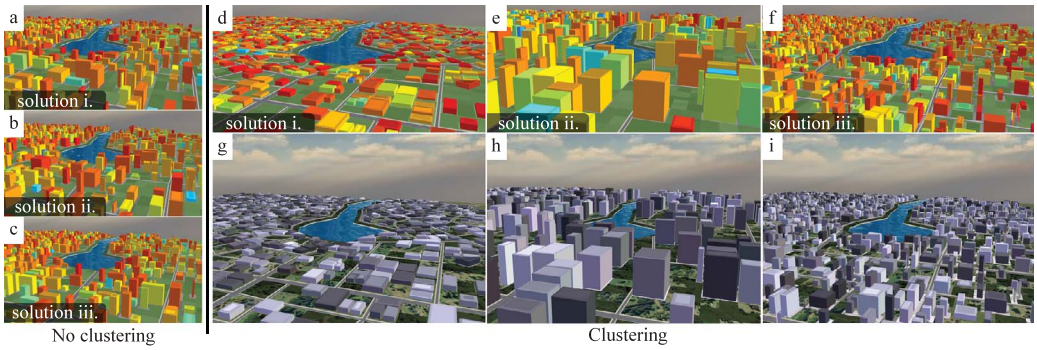

Fig. 5.2. **Variability.** a-c) We show the three top solutions generated by our system, with variability disabled, for a desired sun exposure indicator value. d-f) Next, we enable our solution to increase solution variability and obtain three clearly non-similar top solutions. g-i) Show the corresponding models of d-f but using our interactive rendering engine.

ANN. Figure 5.1 presents a comparison of the measured vs. estimated values by our ANN. Figures 5.1a-c compares the accuracy of our ANN. We run 200 sampled parameter inputs and feed our procedural model and out ANN, we display as scattered point the value measured (x-axis) respect the predicted by our ANN (y-axis) for three different indicator values. Note that the ideal result would be a line (perfect correspondence). Figures 5.1d-e compare the target value vs. the measured one. The result shows that 90% of the samples have a fitting error below 10%.

Variability. Figure 5.2 shows how our solution search method 3.3.4) can help to find solutions that present variability. For this example, the user defines that wants a 3D model with an average sun exposure of 30%. Using the k best solutions (closest to the target indicator value), with $k = 3$ we obtained the solutions presented in Figure 5.2a-c. The solutions have very similar input parameters (they belong to the same cluster) and, thus, they present very similar visual appearance. In contrast, when we use our $k-means$ algorithm, the solutions have the target indicator values but with significant different input parameter yielding 3D models that are signifi-

Fig. 5.3. **Content Design Example.** We show an example of design using a high-level indicator, in this case, an indicator that measures the open/bright vs. compact/dark of a 3D urban model. Using a single slider the user can control it: a) high shadowing or compact, b) medium shadowing, and c) low shadowing or open.

cantly different (Figures 5.2d-i). Using this technique, the user can explore different alternatives but still keeping the control over the target indicators.

Content Design. Figure 5.3 shows a content-design example. The designer wants to control how open/bright vs. compact/dark a city model is. Our method allows the user to do such control just altering one slider. Our system learns how the indicator behaves and which subtle interdependencies of the needed parameter changes (shown in the insets in Figure 5.3) to achieve the desired behavior.

Multiple Land-Use Dependencies. Figure 5.4 shows an example of multiple land-use dependencies. The user implements a new indicator value, *landmark visibility*. This indicator value tries to capture the inhabitants preference of a city to see the main landmarks from their building. We set up a scenario as presented in Figure 5.4a, we have two landmarks on the North, and three different place-types placed in a horizontal layout. We want to be able to control with a single slider how many buildings are able to see at least one of the landmarks. We train our system with this layout. Then the user sets the target indicator value to 15%, 30% and 75%. Our system is able to find such configurations with a 3% error. Note that our system learns the most important parameters that need to be controlled, the building heights and the percentage of parks in each one of the three place-type instances. For low

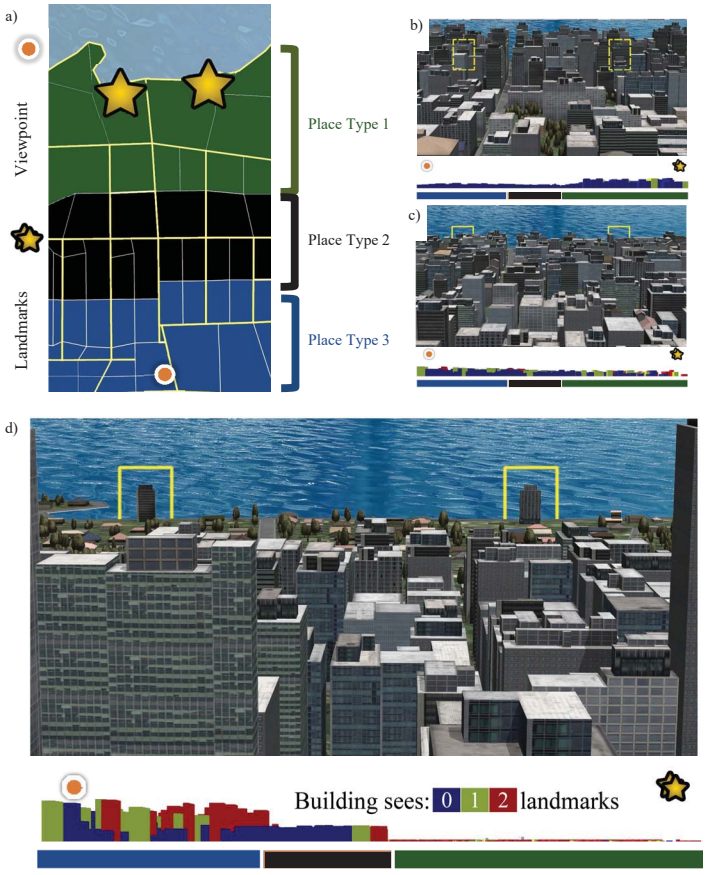

Fig. 5.4. **Global Indicator Control.** This example focuses on a global landmark visibility indicator. a) Top-down view of the city model and user-selected landmarks. b) Initial 3D city model configuration where the landmarks are not visible from most buildings (yellow boxes). c-d) User increases the desired amount of landmark visibility and the system interactively alters the city model. Below images b-d is a color coded profile of the city showing how many landmarks are visible.

landmark visibility, our system sets high buildings close the landmarks (Figure 5.4b) such that just those building do have visibility. For middle visibility, our system learns that needs to set homogeneous heights along the three place-types. Finally, for high landmark visibility, our system propose to set low-rise buildings close to the landmarks (5 floors), mid-rise buildings (15 floors) to the middle place-type, and

high-rise buildings (30 floors) the furthest from the landmark. Note that this might seem an obvious solution when observed, but our system learn it without any user interaction and allows to control it with a single slider.

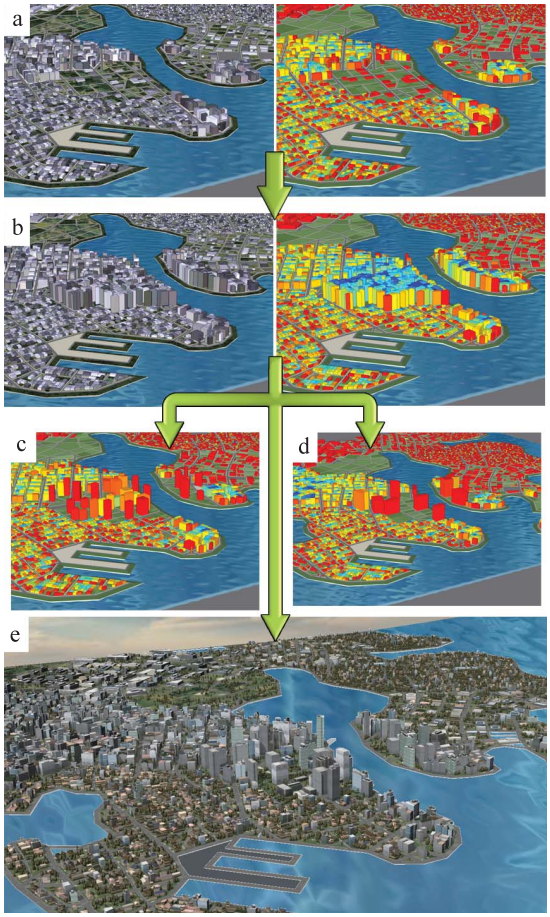

Fig. 5.5. **Local Changes for Global Indicator Control.** a) One of the nine neighborhoods of a city is redeveloped so that the average floor-area ratio of the entire city increases. b) The system proposes a solution that satisfies the target floor-area ratio but that reduces the sun exposure of the area. The user then requires the system to find a solution that maintains a high sun exposure. Three different solutions are produced (c, d, e) that exhibit different styles but satisfy the constraints on both indicator values.

Local Control for Global Indicator. Figure 5.5 shows an example on how we can achieve global control of a 3D urban model (in this case, the whole city counts with nine place-types) just with the redevelopment of one of them, i.e., local control. We set up our city with land-uses as shown in Figure 5.5a and we let our system to learn how to control the global indicator values changing one land-use (the one to redevelop). We use our system to improve the average floor-to-area ratio of the entire city to 5.2. Our system finds such a solution with a 3% error. However, the proposed model has too low sun exposure. Thus, the user uses the sun exposure slider to increase its value. Our system finds several solutions that has the desired floor-to-ratio value and also has the user-specified sun exposure (Figure 5.5c-d). Finally, the user selects one based on style preferences (Figure 5.5d) and exports it to CityEngine (Figure 5.5e).

5.2 Traffic Simulation

We used our framework to create and edit a variety of city-scale 3D models obtained from OSM or our procedural engine having up to 360 km of roads. The simulation is implemented both on the CPU and on the graphics card (CUDA). All rendering is done using our custom shader (see video). All editing is interactive, and results appear while editing. All example sessions were completed in less than 10 minutes (and typically in less than 5).

Example Designs. Figures 5.6-5.8 show the results from several example design sessions. Figure 5.6 demonstrates a local inverse design using downtown Boston (9a) with 1393 streets, 4767 intersections, and a total road length of over 290km. The people and job distribution has been extracted from a GIS. The user first runs the simulation with 110k vehicles. Results show significant traffic crossing the Boston Commonwealth Park and the nearby government buildings. The user then draws two areas to reduce the road occupancy and defines the maximum occupancy to be 20% of those roads. Next, the simulation is run in three different scenarios. In the first

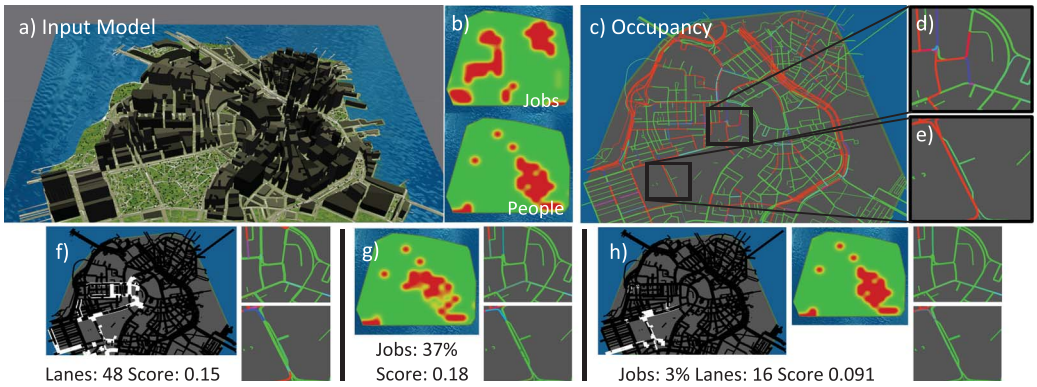

Fig. 5.6. **Local design.** a) Fragment of central Boston. b) Job and people distribution from GIS sources. c) Initial road occupancy as per our traffic simulation. d-e) Close-ups. Three solution options: f) solution by only changing lane directions, g) similar as previous, but only jobs distribution changes, and h) using both change types (best solution).

scenario, the user only allows changes in lane directions (9f). This restriction sparks changing 48 lane directions to reach the desired behavior. Then, the user only allows changes in job distribution (9g); this could be useful to improve traffic with the cost of a company office reallocation. However, it causes 37% of the jobs to relocate. Finally, the user enables lane direction change and job re-distribution resulting in just 16 lane changes and 3% of the jobs moving (9h). Moreover, we reduce the occupancy to just 9% as indicated by the specified traffic zone behavior.

Figure 5.7 represents an interactive session with a global design objective. The user loads the map of Madrid, Spain from OSM (10a). It contains 2288 streets, 6141 intersections, and more than 330km of roads. The user defines a desired traffic behavior for the entire city and for 220,000 vehicles (10b). Our system finds a solution that produces the given traffic zones with just a few iterations (10c-d). This is possible thanks to the tomo-gravity model (this is the only result that uses subsection 3.4.3 initialization). Afterward, the user wanted an alternate traffic behavior of the city (10e). The system was also able to find the model that yields such a traffic behavior (10f-g). This session took under 5 minutes.

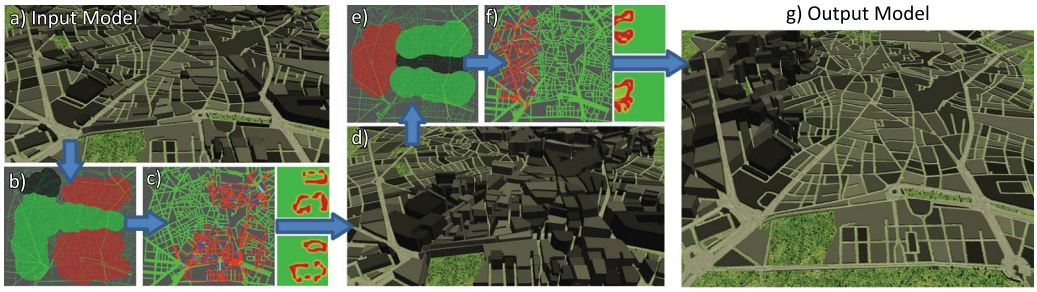

Fig. 5.7. **Global Design.** a) Fragment of Madrid. b) User draws the desired traffic and c-d) our system first uses the tomo-gravity model and then MCMC refines it. e-g) Another editing iteration produces the final output.

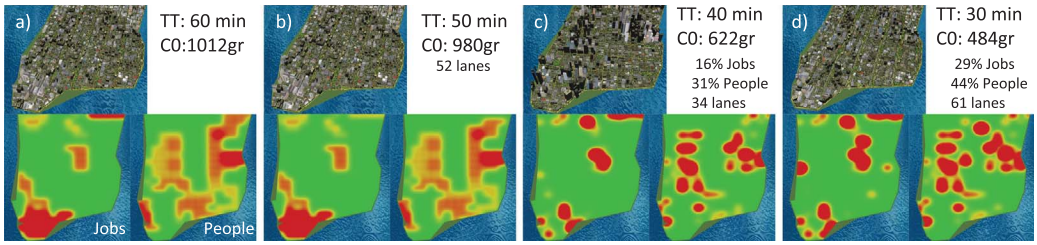

Fig. 5.8. **Global Optimization.** a) Fragment of New York. It has an average travel time (TT) of 60min and CO emission of 1012gr. Our system finds that b) by just changing lanes it is able to achieve the 50min goal. c-d) To reach 40min and 30min, it is necessary to change people, jobs, and lanes.

Figure 5.8 shows a global traffic optimization. We procedurally generate a fragment of New York City using GIS data, producing over 900 streets, 620 intersections, 25k cars, and 360 km of roads. The initial urban model yields an average travel time of 60 minutes and 1012 gr of CO per person (11a). The user defines three desired travel time values (i.e., 50 minutes, 40 minutes and 30 minutes). Our system finds that it is possible to improve the traffic to a 50 minute travel time by changing 52 lane directions (11b). However, in order to achieve a 40 minute travel time, more radical changes are required –16% of the jobs and 31% people should be relocated

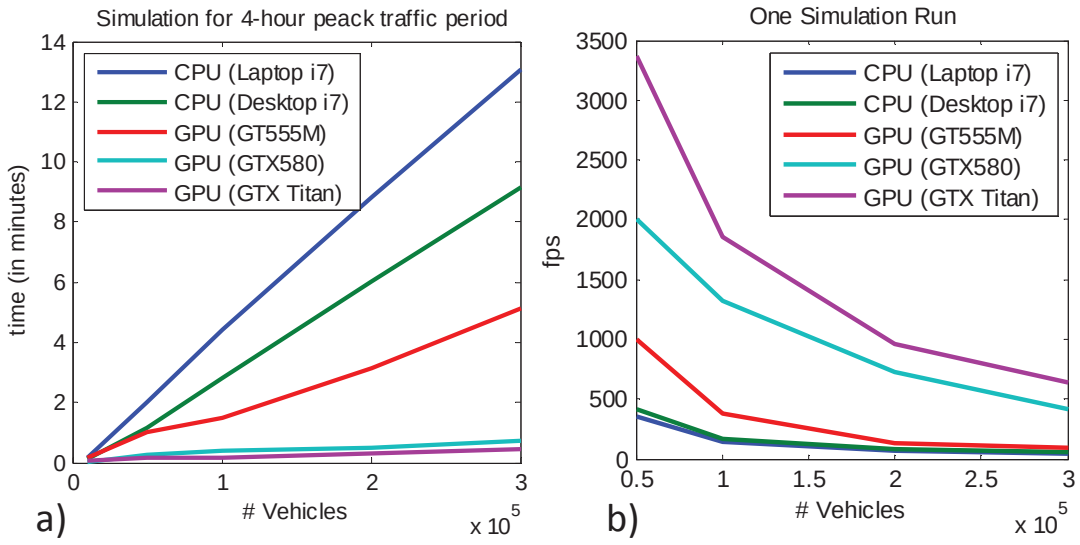

Fig. 5.9. **Performance.** a) Times (in minutes) to perform a 4-hour simulation b) The number of simulation steps per second.

(11c). Finally, to achieve a 30 minute average travel time even more changes are required (11d). Each of these sessions took 10 minutes or less.

Performance. We present a performance summary using up to 300k vehicles on 200 km of roads (with no rendering overhead). Performance is shown for CPU and GPU versions (Figure 5.9a). A four-hour peak traffic period simulation (i.e., 6am-10am) of 300k cars takes as little as 24 seconds, including trip planning, simulation, and estimating occupancy and indicators. Figure 5.9b shows that with 300k cars our system can compute 636 simulation steps per second.

Analysis. Figure 5.11 compares the behavior of our two search strategies for 30k cars (Sections 3.4.4 and 3.4.4). Note that due to the non-linear nature of traffic, solution process is not monotonic. We define traffic zones that reduce traffic through one of the main arterial roads. Further, we set the cost function to be the total number of changed lanes. On the left, the goal-oriented optimization minimizes the objective function without any constraints. The found low-score solution requires

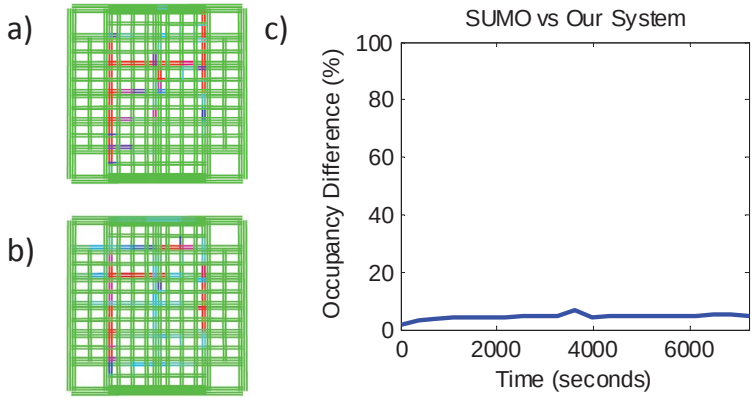

Fig. 5.10. **Occupancy Comparison - SUMO vs. Our System.** Traffic flow measured by our system (a) and SUMO (b); red = complete utilization, green = empty street; c) Error of occupancy measurements over time.

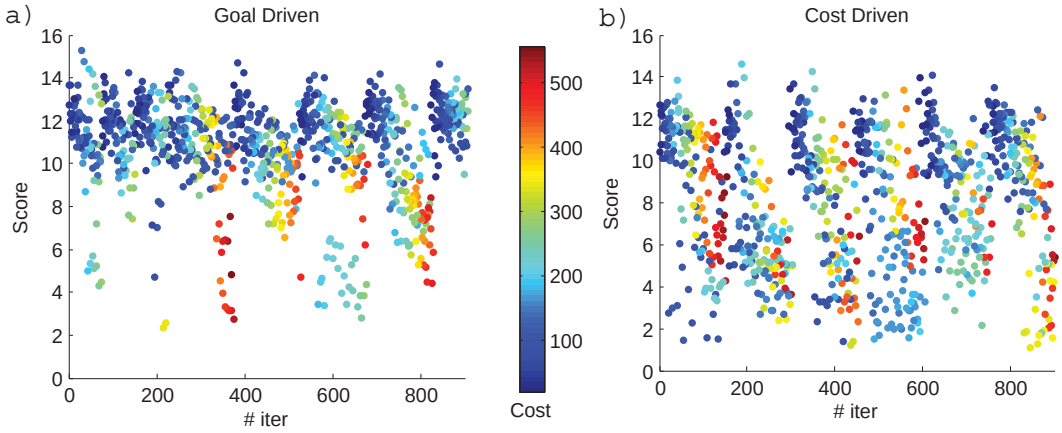

Fig. 5.11. **Search Strategy Comparison.** a) As the iterations pass, goal-driven optimization is able to find solutions of a lower score, but their cost is unbounded. b) In contrast, a cost-driven optimization attempts to reach near the desired score and then it continues but tries to minimize cost.

almost 350 lane changes. On the right, the cost-driven optimization finds solutions with even lower score and with much lower cost (i.e., just 50 lane changes!). This

occurs because the second acceptance formula is more relaxed and thus other state changes can occur.

We compare performance to those of Sewall et al.'s [33] macrosimulation engine and to SUMO's open source microsimulation engine. SUMO's performance, as well as ours, is dependent on the number of vehicles and on the simulation time. Sewall et al.'s [33] method is claimed to be linearly dependent on road network size and on the simulation time. Their largest reported network has 140,000 cars and only 10 km of roads. Their fastest system (8 cores) was reported as 50 seconds which was 54 times faster than their SUMO performance. For a network with the same number of cars but with 200 km of roads, our approach only takes 13 seconds which is 81 times faster than our SUMO performance. The amount of simulated time is not reported for Sewall et al. [33] thus a direct comparison is hard to make. However, if we linearly extrapolate their performance from 10 km to 200 km of roads, our method would be 77 times faster than Sewall et al. Moreover, our method yields disaggregated per-vehicle data. Finally, we use SUMO to evaluate simulation accuracy (Figure 5.10). Our method produces occupancy values that, on average, are within 6% of SUMO-computed values.

5.3 Weather Simulation

We have used our approach to design and simulate weather in various synthetic cities. Our results include a forward design tool, an inverse design tool, comparison, and performance analysis.

5.3.1 Weather Forward Design

Figure 5.12 shows two exemplar cases of the method described in Section 2.6.4. The user draws the distribution of land use (Figure 5.12a) and designs the procedural model of the city (Figure 5.12b). Then, the user selects some initial procedural conditions weather conditions (top) and selects a location and date and loads real

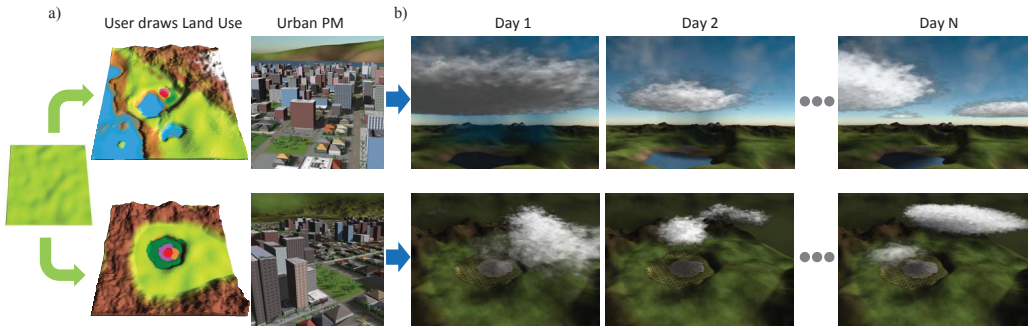

Fig. 5.12. **Forward Design.** Two examples of forward design. a) The user draws the land use distribution and uses our Urban PM to design a city; b) the simulation runs indefinitely for this procedural model, we display different days of a year.

sounding (bottom), in this case from Miami, FO. Finally, our model is able to simulate any number of days of realistic weather (Figure 5.12c).

5.3.2 Weather Inverse Design

We demonstrate our weather design tool in Figure 5.13, Figure 5.15, and Figure 5.16. We show how our framework can be used to design, control, and optimize different scenarios.

Cloud Design.

Figure 5.13 shows an example of the use of our high-level design tool. The user wants to control the cloud coverage (percentage of sky covered by clouds) of a procedural city throughout a day. To compute the cloud coverage, we accumulate the amount of clouds computed for our Radiation Model (Section 2.6.3) and average over the domain space. The user interactively paints an urban and non-urban area (Figure 5.13a). Then, the user designs the weather behavior (Figure 5.13b) defining as objective the mean percentage of cloud coverage ($\bar{\sigma}$) and, optionally, the permissible error (this allows the user to define an admissible range) of cloud coverage ($\check{\sigma}$) for m different time ranges r. This defines an objective cloud coverage set $\{\bar{\sigma}_{r\in r_i}, \check{\sigma}_{r\in r_i}\}$

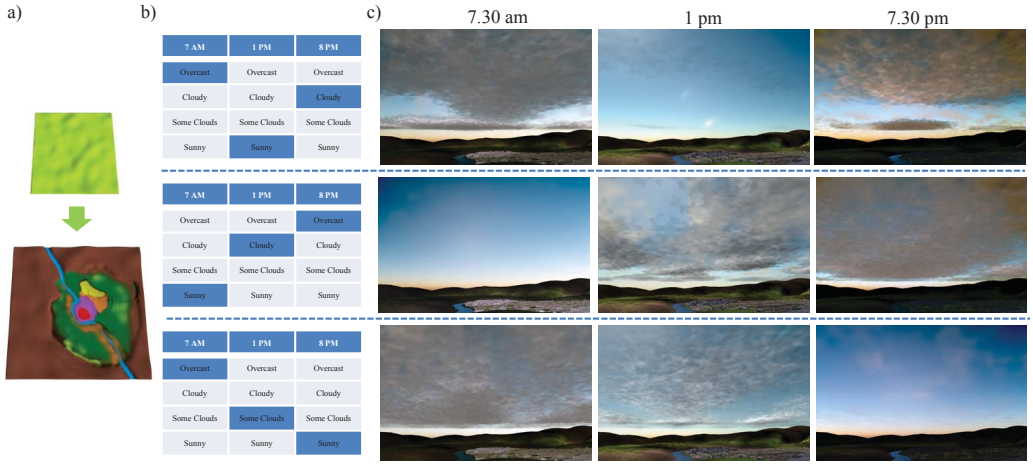

Fig. 5.13. **Cloud Design.** Two examples of cloud design. a) The user interactively draws a land use distribution; b) the user selects the high-level behavior of the weather; c) the system finds such weather and the weather sequence is visualized.

where $i = [1, m]$. Then, we run the optimization to find the closest solution (Figure 5.13c).

For this example, we fix ω_l and ω_p since we do not want to alter the 3D model. To alter ω_w, we set-up the optimization: i) to use our multi-seed approach, ii) to set the variable ranges within plausible physical values, iii) to just alter the control points of U and q_v, iv) to use $\eta = -\infty$ (error function optimization). Finally, we define the error function to minimize the computed cloud coverage respect the user-specified one as

$$E(\Omega) = \sum_{i=1}^{m} \max\left(0.0, |\sigma_{\Omega r \in r_i} - \bar{\sigma}_{r \in r_i}| - \check{\sigma}_{r \in r_i}\right) \tag{5.1}$$

To speed up the process, the user can decide to use our 2D XZ simulation model. Our 2D XZ model uses the same formulation and code than for 3D but drops one-dimensionality (in this case the depth, Y) to create a fast approximation. We use it here as an approximation to compute the cloud coverage. Figure 5.14 shows the error for three different scenarios of the computed cloud coverage using our 2D XZ and

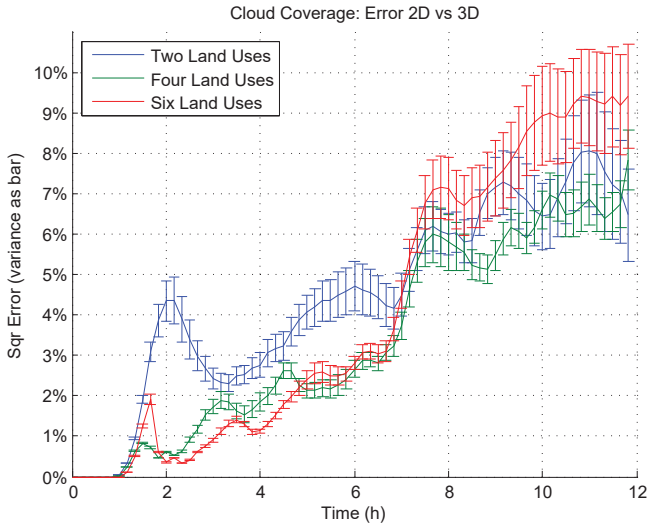

Fig. 5.14. **Difference 3D vs 2D simulation.** We show the square error of the cloud coverage of three different scenarios when it is simulated using our 2D and 3D simulator for 50 different initial conditions.

the complete 3D version. As can be observed the error for time periods of 12 hours have a maximum error of 5% and an average of 3% that provides enough accuracy for the task in hand. The presented solutions ware found using 12 initial seeds, 4 steps, and required less than 5 minutes. Optionally, the user can then explore the rest of evaluated solutions (plotted as temporal cloud coverage changes) and select a more suitable simulation and visualization.

Rain Design.

Figure 5.15 presents an example of our inverse modeling tool. We use our system to control (increase and reduce) the rain of the city through the change of the non-urban land use. We constraint this example because we can argue that it is very expensive (economically) to change the land use inside urban areas; however planting trees (230$ per acre) or changing the land use to crops, it can be something that can be explored, even as a temporal solution. Note that such considerations are being

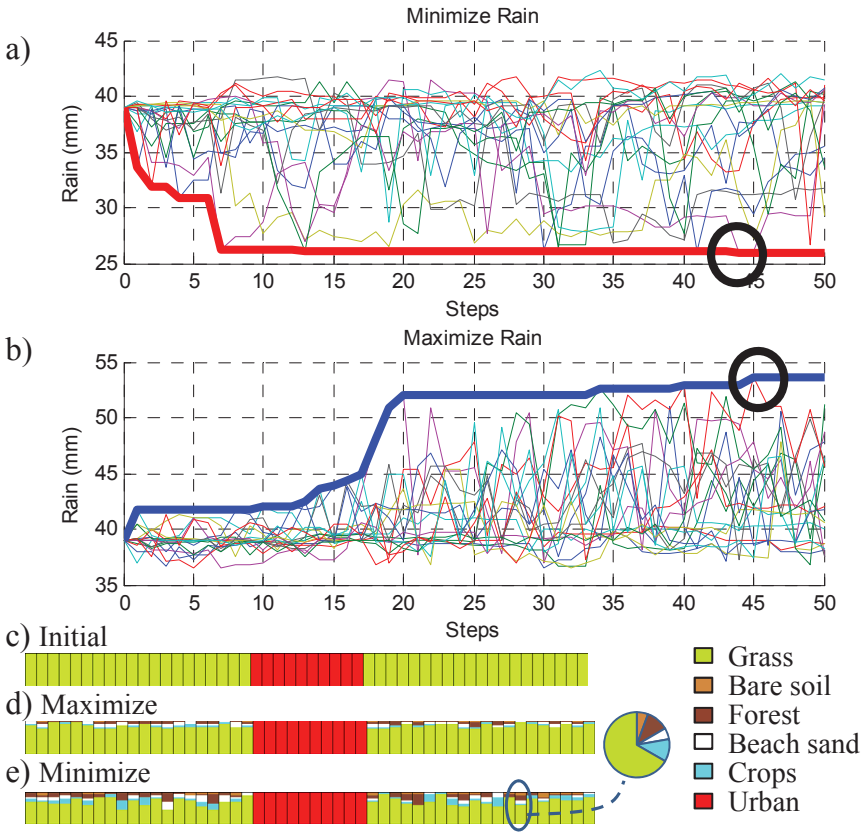

Fig. 5.15. **Rain Design.** We show the result to optimize the rain levels in a city, to increase it a); and to decrease it b). We overlap the result for each energy level and highlight the minimum/maximum value found so far. c) shows the initial land use distribution of the center cross section. d-e) are the best solution for each optimization.

actively studied [139,140]. To limit further the solution, we define that the maximum percentage of land use change of each grid-cell to be 50%.

Note that increasing the rain rates of an area can be interesting for many reasons. An example would be to alleviate seasonal droughts or flood potential [141]. Decreasing rain can be useful for humid and rainy areas (e.g., Seattle, WA) or even to palliate the spread of insects such as mosquitoes. A wide range of environmental services have

been envisaged for urban greening, and optimization tool such as discussed here is critically required.

For this example, we fix ω_p with the default procedural parameters and ω_w with a warm summer day. To control the rain, we alter ω_l of the non-urban areas (green in Figure 5.15c) allowing changes to 'bare soil', 'forest', 'beach sand', and 'crops'. We use our system with one unique seed (the original scenario) and run it with 20 chains with different energy levels $\beta \in [10, 650]$ and 50 steps. In Figure 5.15a-b we present the evolution of rain for each chain, we highlight as a thick line the minimum/maximum value found so far for that number of steps, the circled state is the one selected as optimal.

To compute the total rain fallen in h hours, we use the concept of *rain intensity*. Each s min of simulation we sample the rain intensity (mm/h) in each bottom level grid-cell. We use the classic [Marshall and Palmers 1948] derivation to compute the rain intensity as $RI = 360\rho_0 V_r q_r$ where terminal velocity is $V_r = 3634(\rho q_v)^{0.1354}$. The total rain is computed as the average of RI per hour times the elapsed time (in our case 12 hours); i.e.,

$$E(\Omega_t) = \sum_{t=0}^{h} \frac{60}{s} \sum_{i=0}^{grid_x} RI(t, i) \tag{5.2}$$

Using this formula as the objective function, rain is minimized (5.15a). In order to use the same function but as maximization (5.15b), we invert the terms in the acceptance ratio $E(\Omega_t)/E(\Omega_{t+1})$ instead of $E(\Omega_{t+1})/E(\Omega_t)$. As seen in the particular case of Figure 5.15, we achieve a rain decrease of 45% (Figure 5.15a) and an increase of 35% (Figure 5.15b). Note that such 'drastic' changes are indeed possible since we included 'desert' as a land use category. This land use category has a very high albedo. In contrast, if we eliminate this option, the maximum decrease is only 19% and the maximum increase is 16%.

Temperature Design.

Figure 5.16 presents an example of our cost minimization. We use our system to reduce one degree of temperature in the city center (i.e., $\eta = E(\Omega_0) - 1C.$) so as to alleviate the higher temperatures produced by the urban heat island effect. Reducing

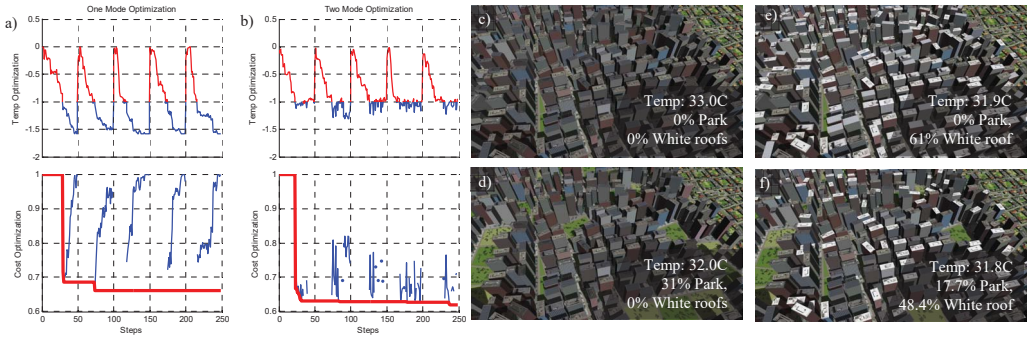

Fig. 5.16. **Temperature Design.**a-b) We show the behavior of a biased optimization for example e) of this figure; a) if just one mode is used (optimize temperature); if we use the two models optimization (optimize temperature and cost); c) the original model; d) altered model that achieves one degree reduction by introducing more parks; e) alternative model that achieves the same goal but uses white roofs to increased albedo; and f) a solution with both parks and white roofs (note the reduction in both).

the urban heat island by only a few degrees is recognized as a difficult though valuable goal [70]. For example, super dense cities such as Beijing can benefit from reduced heat islands that would decrease pollution [142].

As allowable changes to the initial model, we fix ω_l and ω_w and explore three alternative options for ω_p: change some urban land to parks, paint some roofs to be of high albedo (e.g., White Roof Project [143,144] is currently promoting this option), and perform both of the aforementioned changes.

First, we want to compare the standard acceptance ratio where one variable is optimized (Figure 5.16a) with our modified formula where we optimize a variable while we keep the cost to the minimum (Figure 5.16b). In Figure 5.16a, the temperature is progressively reduced by the optimization method until it crosses the desired goal just once per energy level. After it crosses, the temperature is reduced further, but now the cost may increase. The best solution is found in one of this single crosses. On Figure 5.16b, we present our cost minimization method, the temperature is reduced by the optimization, as before, but when the desired level is achieved ($\eta = E(\Omega_0) - 1C$),

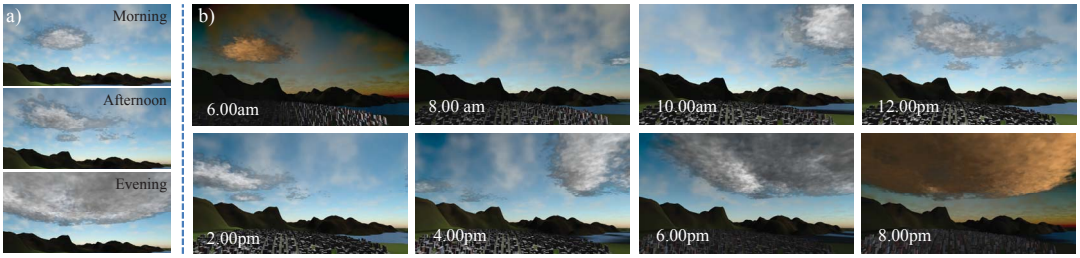

Fig. 5.17. **Global Design.** a) The user selects the desired cloud states. b) Our system is able to simulate the user-selected behavior.

the optimization minimizes the cost. Both behaviors alternate trying to find the goal temperature with the minimum cost. Figure 5.16c shows the original model. Figure 5.16d shows the model with 31% more parks and $1C$ degree reduction in temperature. Figure 5.16e shows an alternative option with 61% more white roofs and $1.1C$ degrees lower temperature. Figure 5.16f uses both more parks and white roofs to achieve $1.2C$ degrees reduction. Overall, the design tool enables quick exploration of options to achieve the desired weather/climate change.

Global Design.

Figure 5.17 shows an example of our solution towards global design. The user selects few interesting cloud states (Figure 5.17a) from all the sampled precomputed states (in our case, from $128 \cdot 10 \cdot 72$ states), the order, and the time period of simulation. Our system, changing the boundary conditions, is able to simulate using our phisically-based method the user-defined behavior (Figure 5.17b).

5.3.3 Comparison

To validate our physically-based, simplified, super real-time simulator we compare it to state-of-the-art results and systems. We validate each of the three main components of our framework separately. One option to validate the first component, our fundamental equations, is to perform a comparison to well-known benchmark cases

of a bubble of cold air and a bubble of warm air under prescribed conditions [145]. Figure 5.18 and 5.19 have 2D visualizations of such bubbles. In Figure 5.18a, an elliptical cold bubble ($-12.5C$) is placed in the center of the domain at a height of $3km$, using a grid cell size of $0.125km$ and a time step of $0.25s$. As should be the case, after $450s$ of simulation the bubble sinks and Kelvin-Helmholtz eddies are produced (Figure 5.18b). We impose wind of $20m/s$ which as per Wicker and Skamarock [146] makes the test more stringent. At $900s$, as expected the eddies have completed a half revolution around the domain and approach each other (Figure 5.18c). This behavior is very similar to Figure 2 of Wicker and Skamarock [146], a classical reference. Figure 5.19 has a spherical warm bubble of $28C$ and at $2km$ above the ground, and same grid cell size and time step as in Figure 5.18. After $900s$, the bubble rises as should be the case as per Ahmad and Lindeman [147] – no wind is introduced in this benchmark case.

To verify the second component, we compare our clouds and precipitation model to the WRF-ARW (Weather Research and Forecast Model Advanced Research Version) and an expected cloud/rain formation process. Figure 5.20a-b show the computed value of water vapor q_v at different heights in a rural area and in an urban area using WRF and using our model both models behave similarly. However, WRF does not model clouds explicitly nor render them. Thus, we look to the literature on cloud dynamics. Cumulus clouds are formed by convergent-divergent zones producing powerful updrafts that form large vertical clouds and typically significant rainfall. We use our simulator to generate such a cumulus cloud adding a $3C$ warm bubble in the city center and successfully show its three main stages (Figure 5.20c-e): towering cumulus stage, mature stage, and dissipating stage. As expected, this large cloud forms a thunderstorm during the third stage that resembles a classic supercell [148].

For the third component, we compare WRFs radiation model to our simplified implementation of radiation model. The simulated domain consists of a simple circular city in the middle of an otherwise green terrain. As can be observed in Figure 5.21a, the relative temperature difference between urban and non-urban areas, defining the

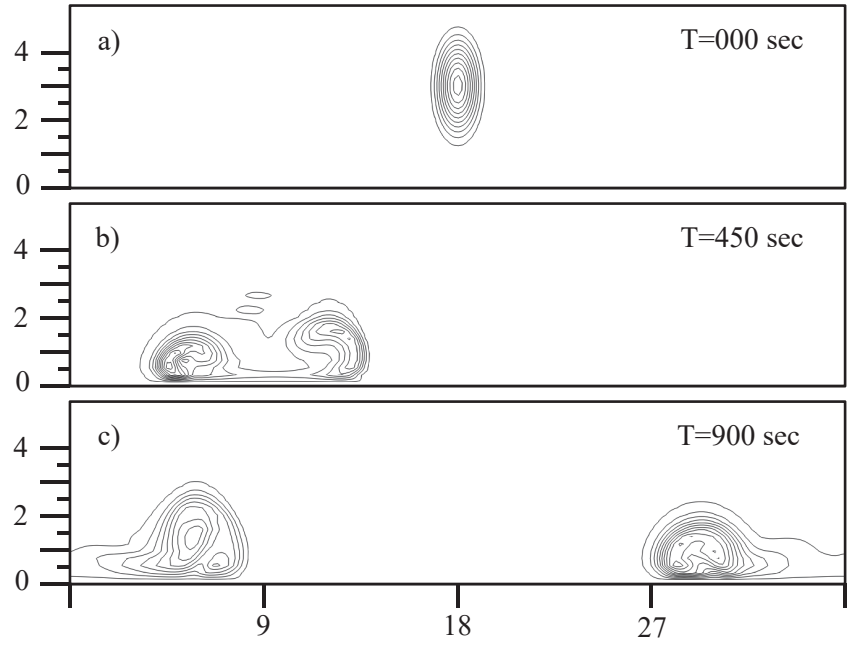

Fig. 5.18. **Cold Bubble.** Our evolution of potential temperatures similar to that of Wicker and Skamarock [146]. Vertical axis is height and horizontal axis is spatial x-axis location (both in km). Simulation isolines at a) 0 seconds, b) 450 seconds, and c) 900 seconds.

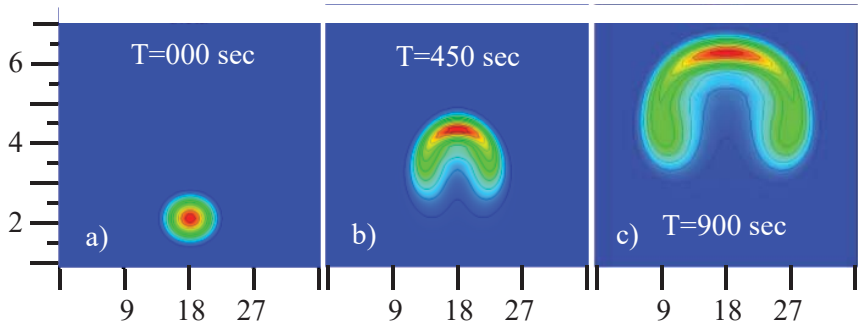

Fig. 5.19. **Warm Bubble.** Our simulation produces potential temperatures very similar to Ahmad and Lindeman [147]. Axes and temporal sequence same as in Figure 8.

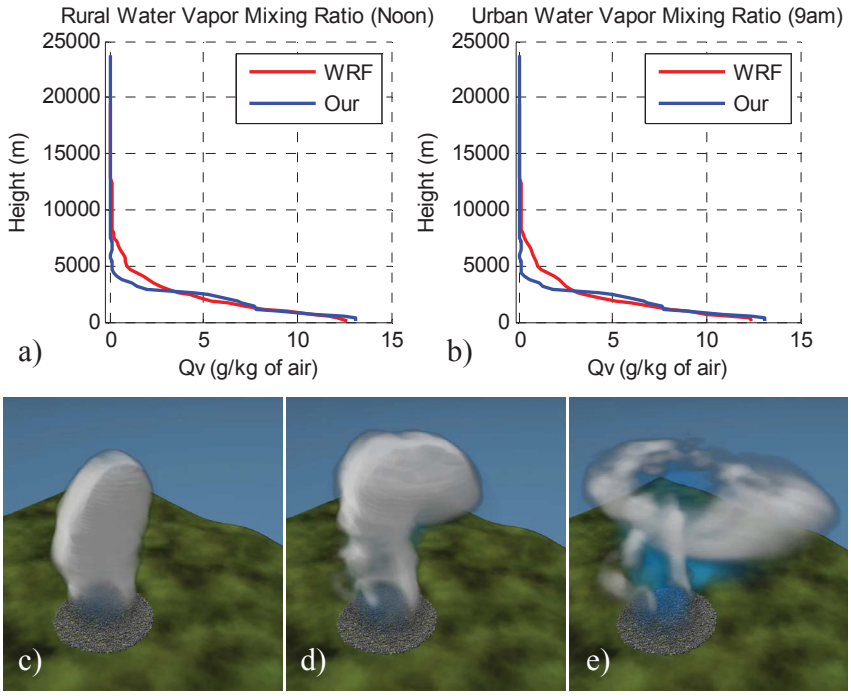

Fig. 5.20. **Cloud and Precipitation Comparison.** a-b) We show the water vapor mixing ratios for an urban and rural area using WRFs and our Kessler Microphysics implementation. c-e) A temporal evolution of a cumulus cloud with our implementation.

urban heat island, computed by the two models is almost identical. The shown curves are the difference between an East-West slice through the middle of the domain and an East-West slice through the southern part of the domain (not intersecting the city) as computed by each model. We also show the temperature evolution over time for a point in a rural area and in an urban area (Figure 5.21b-c), using both models. Our radiation model again behaves very similarly to WRF.

It is important to highlight that, weather forecast are inherently nonlinear and by definition chaotic [149]. Small changes to initial conditions (either prescribed or computed/interpolated by different model routines) can result in differences in the model output. Even for two sophisticated models or for two different physics options

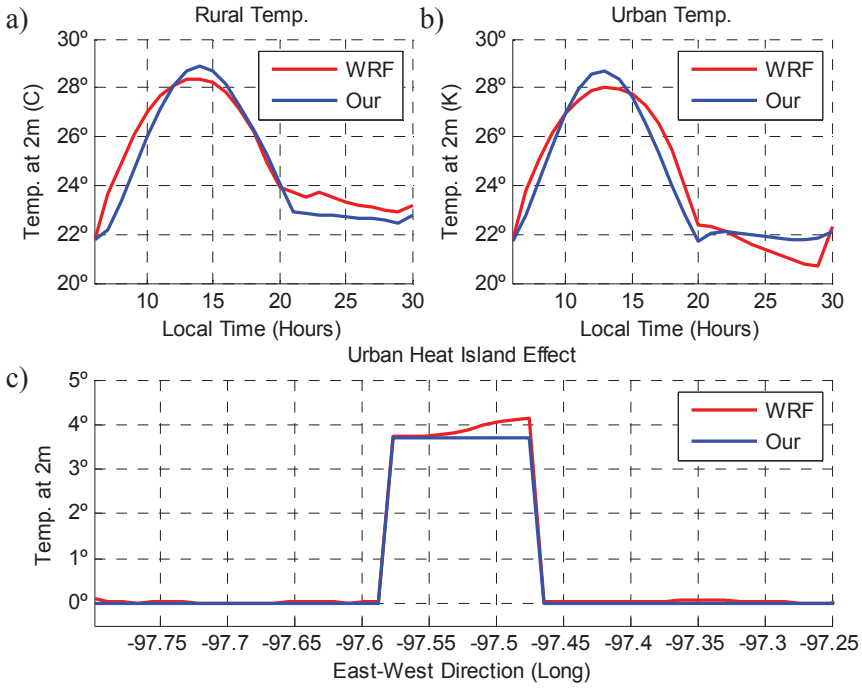

Fig. 5.21. **Radiation Model Comparison.** We compare WRFs radiation model to our radiation model (RM): (a) urban heat island effect as the temperature difference between an urban and non-urban 1D slice using each model; b-c) we show the temperature evolution over time for a point in a non-urban and an urban portion. Our model behaves quite similar to WRFs.

runs the model output can have significant differences. This is the basic of the so-called ensemble weather forecasting [150] used by the meteorological community (and yields values such as probability of precipitation). Therefore, the differences seen in the simulator and the more sophisticated state-of-the-art WRF model is expected.

Performance.

For a simulated volume of 50x50km and 25km high (divided into 50x50x56 grid cells), our 2D XZ system (Section 5.3.2) computes 81 minutes of weather in one second and our 3D system computes 1.7 minutes of weather in one second. In all cases, we use a simulation time step of 1 second. Table 5.1 summarizes our performance on

Table 5.1.
Performance. CPU uses 4 cores while GPU uses 2304 cores. See main text for additional details.

		Time per Time Step (ms)		Faster than Real-Time
		CPU	GPU	(GPU)
2DXY	Fund.	25.02	0.08	12195x
	+Clouds	62.98	0.16	6135x
	+Rad.	64.72	0.21	4878x
3D	Fund.	250.23	5.04	199x
	+Clouds	639.11	9.71	103x
	+Rad.	671.72	9.90	101x

the aforementioned CPU and GPU. Our GPU implementation reaches over 101x with respect to real-time, and it is 68x faster than the CPU counterpart. As a reference, WRF is just 7.1x faster than real-time running on a computer server with 4 AMD Opteron 6176 12-core processors (450 GFLOP) this maps to approximately 1.2x faster than real-time using our Intel Core i-7 (170 GFLOP). Note that this comparison is not straightforward since WRF is optimized to run on servers. For instance, a 'typical' scenario for WRF would be to run a 4 nests of 72x72x48 to simulate 30 days using a high-performance cluster (with 256-512 nodes).

5.4 Geometric Assets and Visualization

In this section, we present the results for our two methods to reconstruct 3D urban models.

5.4.1 Volumetric Reconstruction and Surface Graphic Cuts

We have used our method and system for several large urban examples. Figures 5.22- 5.33, show our results and analysis. Our example dataset consists of a grid of about 58 by 19 aerial viewpoints over central Boston, MA (USA). At each viewpoint, a camera cluster takes 5616 x 3744 resolution images in five directions: one direction straight-down, and 4 diagonally downward facing directions at about 90-degrees from each other when projected on the ground plane (note: our method makes no assumption about the spatial and angular distribution of the camera views). This totals 4667 images from pre-calibrated viewpoints. The area has 1785 buildings assumed to lie on a flat ground plane. We set the default initial building height to 35 meters (assumed residential zone height). Medium-height high-rise zones are set to an initial building height of 125 meters and tall high-rise zones are set to initial overestimated building height of 250 meters.

There are two user parameters, voxel size r and texture size per voxel S. As described before, r defines the voxels size and we found empirically $r = 2$ or $r = 4$ is a good balance in time and reconstruction accuracy. The parameter S can be calculated from r to use the maximum resolution of the images (user can decide to decrease it to speed up the process).

There are two building height clustering parameters: the threshold to discard the column variability and c_e which defines when to stop the clustering process. In our examples, the first parameter is set to two times the standard deviation and the latter to 0.3.

Finally, there are two more parameters regarding the surface graph-cuts that depend on how much the images overlap. In our case, the amount of overlap between patches and the overlap region between building and ground textures are both set to 4m. Reconstruction time depends mostly on the voxel size r and subdivision factor S. For our dataset, a half resolution reconstruction (e.g., $r = 4$ and $S = 4$) takes 22 seconds per building on average (10 hours total time). A full resolution

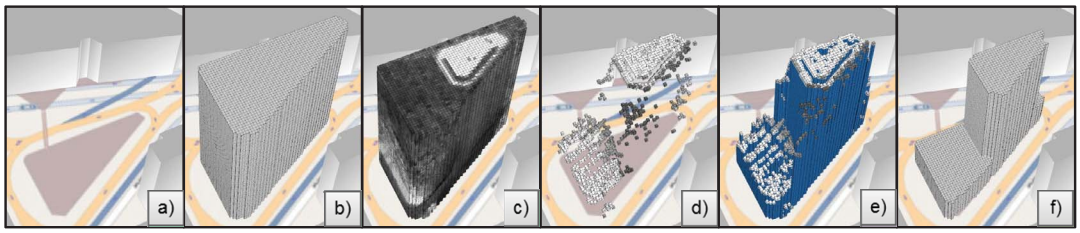

Fig. 5.22. **Volumetric Reconstruction Pipeline.** We show example images from a volumetric building reconstruction. a) OpenStreetMap input image. b) Voxelized-version of the extruded building footprint. c) Per-voxel weighted photo-consistency variance (white = low variance). d) Selection of per-column voxel with lowest variance. e) Vertical support added beneath each per-column selected voxel. f) Final proxy after clustering and filtering.

reconstruction (e.g., $r = 2$ and $S = 4$) consumes 109 seconds per building (51 hours total time). The timing includes local file I/O. A typical building has from 15 to 130 contiguous patches (of the same image id) before graph-cut application and 74 patches on average. A representative building graph has about 150k vertices, 300k edges, 80k triangles before grouping voxels for rendering and 5k triangles after grouping voxels. The ground graph is at pixel resolution and the integrated ground graph-cut solution is stored in a grid of 4x4 12MP images (so that the 16 tiles can fit in texture memory and leave space for building texture atlases).

Memory requirements depend on the stage in the pipeline. Building geometric reconstruction requires about 100MB and can be reduced to less than 1MB per building after processing. Per building graph-cut processing requires less than 200MB and the atlas creation requires less than 850MB (the requirement is higher since the images are loaded at maximum resolution).

Building Reconstruction

We show in Figures 5.22- 5.26 several examples and comparisons for individual building modeling. Figure 5.22 contains intermediate results from the volumetric reconstruction process of an example building. Figure 5.22a has a close-up of the OSM street map used as input. Using an image processing algorithm, we find the building outline and choose a default medium high-rise height in this zone. In Figure 5.22b, we show the initial volumetric approximation subdivided into voxels. Figure 5.22c shows the calculated per-voxel variance it is computed for all voxels but only the exterior voxels are visible. Nevertheless, the photo-consistency of the upper roof structure is evident. Figure 5.22d shows the voxels with minimum variance per voxel column, which begins to reveal the building structure. In Figure 5.22e, we also draw all the voxels beneath each selected minimum variance voxel. Finally, Figure 5.22f shows the proxy model after clustering and filtering. This same process is repeated for all buildings.

In Figures 5.23a-d and 5.24a-c, we show the initial volumetric approximation, the computed proxy model, and the textured result after surface graph-cut processing. Our approach is able to produce reasonable proxies for this variety of building shapes. For comparison, we show in Figure 5.23e the ground truth (obtained by manual modeling) and in Figures 5.23f-h the accuracy of several reconstructions is compared to ground truth using Hausdorff distance: we show the reconstruction error of our proxy (8f) and two versions of space carving (8g-h). As one can observe, the reconstruction error for our proxy is small. To create the first version of space carving, we use Graph-Cut Segmentation [151] (as explained Figure 5.25) to automatically segment the foreground (i.e., the building in view) from the background (i.e., everything else). For the second version, we manually perform the segmentation using a painting tool – a task that it is impractical for large-scale urban reconstruction (e.g., it took between one hour to two hours to create the 25-50 masks of each building). Nevertheless, despite perfect segmentation we found that in general the obtained building recon-

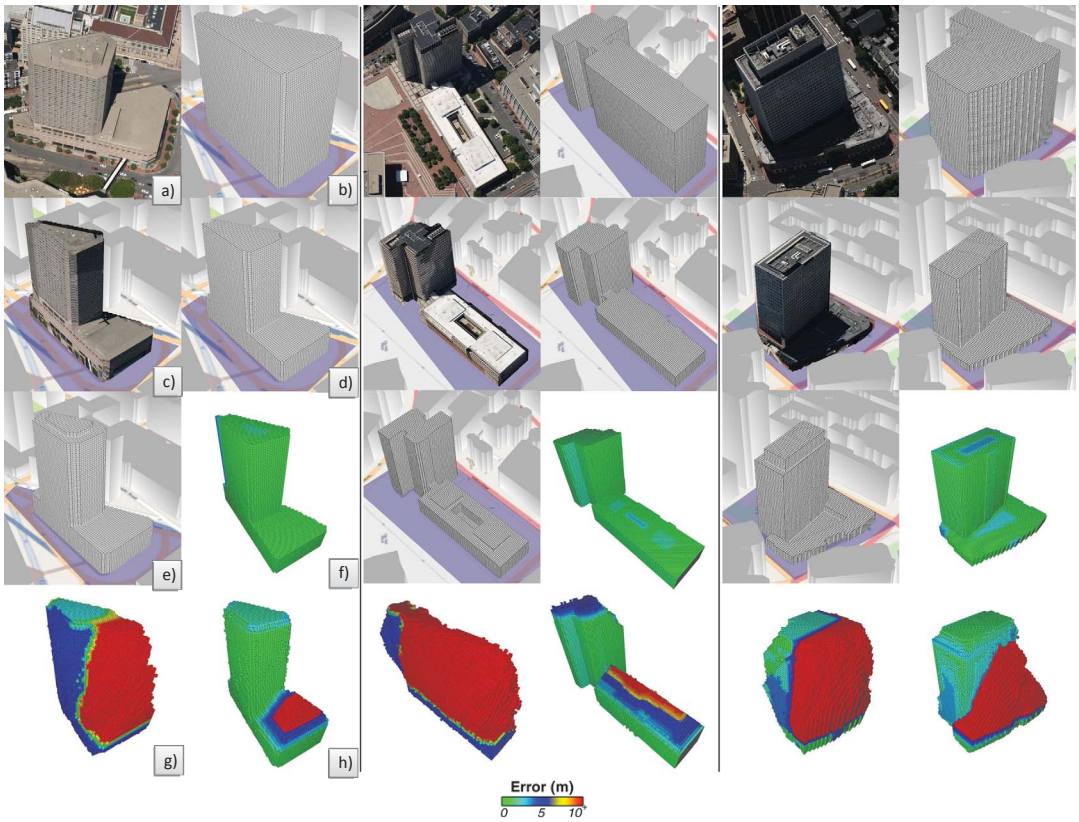

Fig. 5.23. **Building Graph-Cuts and Space Carving.** a-d) Aerial picture, initial voxels, our textured result, and our calculated model with no textures. e) Ground truth and f-h) show Hausdorff distance (color map: green=0m, blue=5m, red=10m or more) between ground truth and our proxy, graph-cut space carving, and manual-segmentation space carving (see text).

struction was inferior to ours. This is due primarily to the relatively sparse image sampling of each building and to the camera viewpoints being above the city (e.g., a distant camera would theoretically see the building more from the side but the view is most likely be occluded by another building).

In Figure 5.26, we present the reconstruction for buildings of different sizes and complexities. Figure 5.26a is a small building of 20m height, 10b is a medium size building of 90m height, and 10c is a large building of 180m height. For each building,

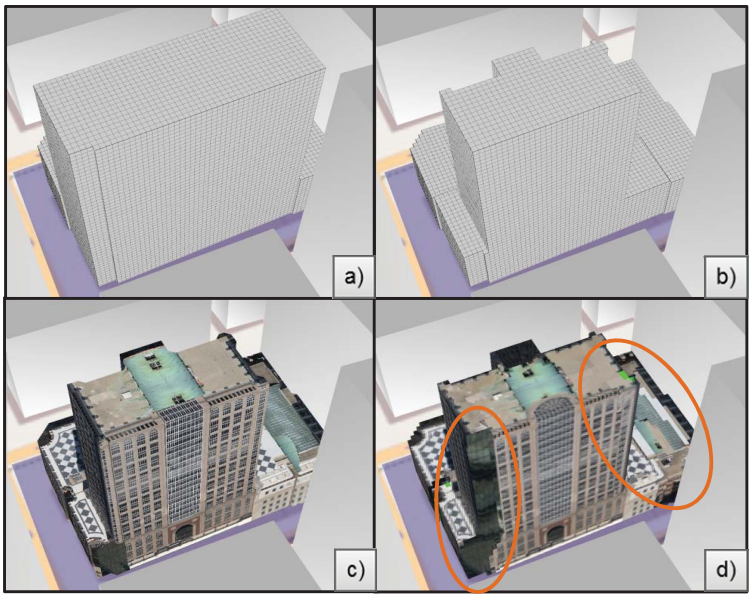

Fig. 5.24. **Texture Mapping Comparison.** a) Initial model. b) Calculated proxy model. Mismatch/discontinuities occur due to geometry/calibration errors that are in general unavoidable in a dense city. Yet, c) our surface graph-cuts compensate for inaccuracies and produce a continuous/coherent texturing, better than d) standard projective texture mapping.

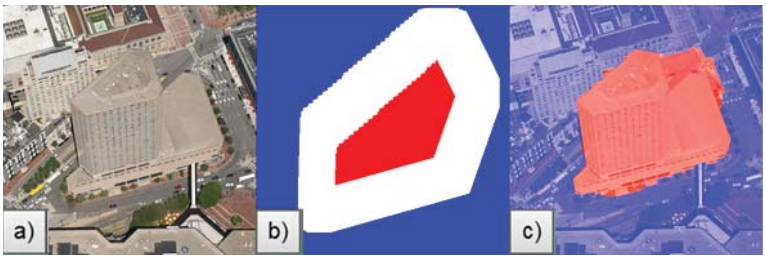

Fig. 5.25. **Graph-cut Space Carving.** To perform space carving, as in Figure 5.23g, we use a) an initial image, b) perform automatic labeling (using the initial voxels as masks), and c) calculate a graph-cut segmentation.

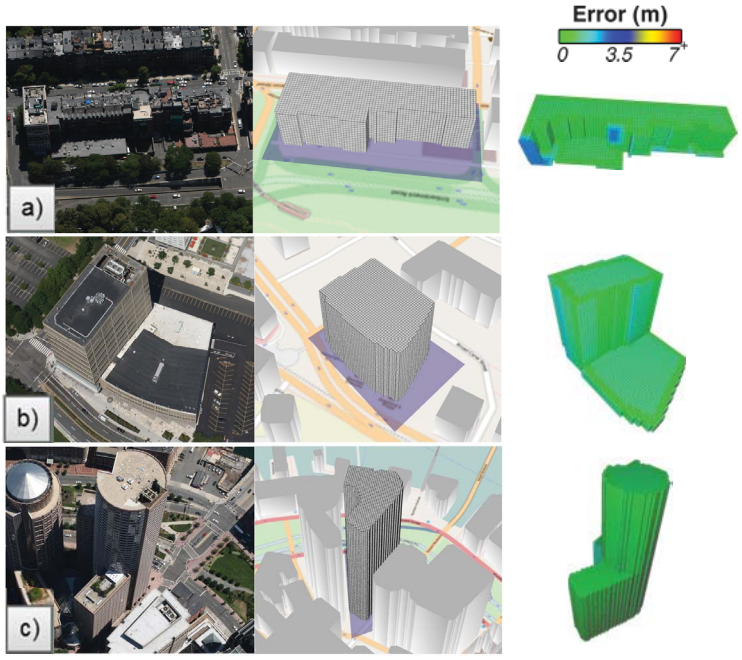

Fig. 5.26. **Building Reconstruction for Various Building Sizes/Complexities.** For a) small building (20m), b) medium size building (90m) and c) large building (180m), (left) aerial images, (middle) initial voxels, and (right) reconstruction error using Hausdorff distance (green=0m, blue=3.5m, red=7m or more).

we show its picture, the initial proxy, and the Hausdorff distance between refined proxy and ground truth. The absolute reconstruction error is approximately constant regardless of the size of the building although the defects are more visible in the small buildings. The error would, of course, be larger if there are not enough images that capture the building.

Figure 5.27 shows the impact of voxel size r in the reconstruction process. When the voxel size is too big, our method is not able to reconstruct the building. When the voxel size is small, the vertical sampling is dense enough to find low variance points and the reconstruction can be performed. However, if the value is too small, excessive processing might occur.

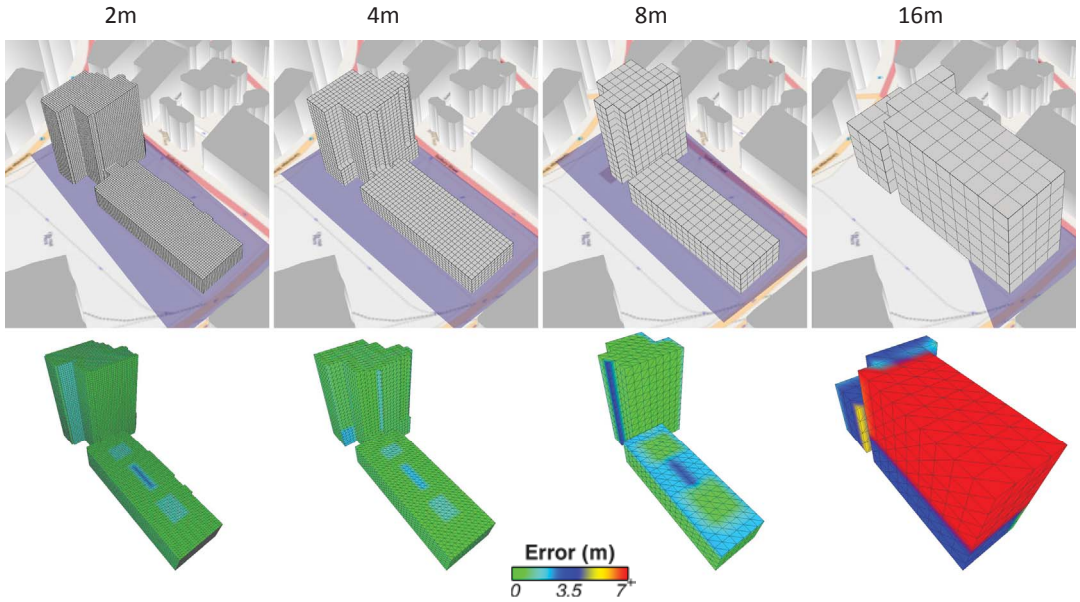

Fig. 5.27. **Result Comparison of Different Voxel Sizes.** From left to right, we increase the voxel size. When the size is too large, reconstruction fails. When the size is small, the reconstruction presents similar results but excessive processing might occur. Hausdorff distance error: green=0m, blue= 3.5m, red=7m or more.

Figure 5.28 summarizes the error in reconstructed building height as compared to ground truth (gathered from Wikipedia) for 15 well-known buildings. The average initial height error is 72%. Our system reduced the building height error to an average error of 1%-3% with a 95% confidence interval.

Surface Graph-Cuts

The impact of our surface graph-cuts is observed in Figures 5.24c-d, Figure 5.29, Figure 5.30, and Figure 5.31. Figure 5.24d contains the result of a nave projective texture mapping. The imprecision in the proxy model, camera calibration, and the high-level of occlusion with neighboring buildings makes it challenging to obtain a perfect texture-mapping. Our additional use of (multiple) building surface graph-

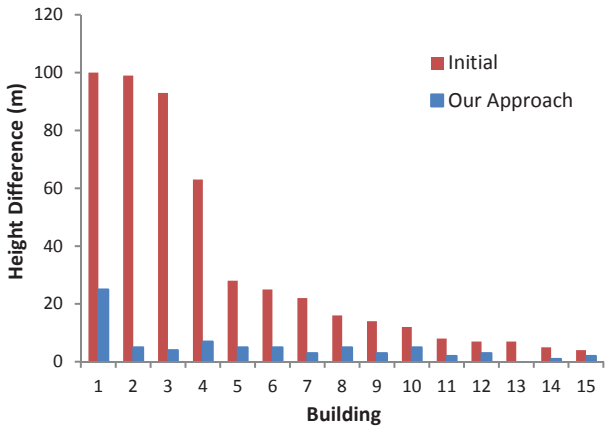

Fig. 5.28. **Reconstructed Building Height vs. Ground Truth.** For 15 buildings, red bars represent the difference between the initial model height and ground truth. The blue bars indicate the difference between our refined model and ground truth.

cuts is able to compensate for these imprecisions and produce a visually-plausible approximation to the buildings appearance (Figure 5.24c).

Figure 5.29 contains a comparison of our graph-cut algorithm with projective texture mapping over the proxy. We compare the original building (middle) with two altered proxies to see how the proxy error affects the texture step. To create the altered proxies we expanded the original building in all directions of the building +10% (left) and we collapsed in all directions of the building -10% (right); in both cases we added a random noise of 5m in the height map. As observed in the top row, our approach manages to make less visible the error in the transition in the top images. Moreover, our approach compensates for the incorrect proxy and is able to eliminate the unwanted appearance of content (e.g., sidewalk, bushes, and side walls). In this example, it is accomplished by automatically extending the wall texture to meet the roof texture, thus producing a transition with reduced visual artifacts however, the solution while smooth might not be physically correct. Our technique cannot always produce an improvement (i.e., compare bottom right picture of 14c with the bottom

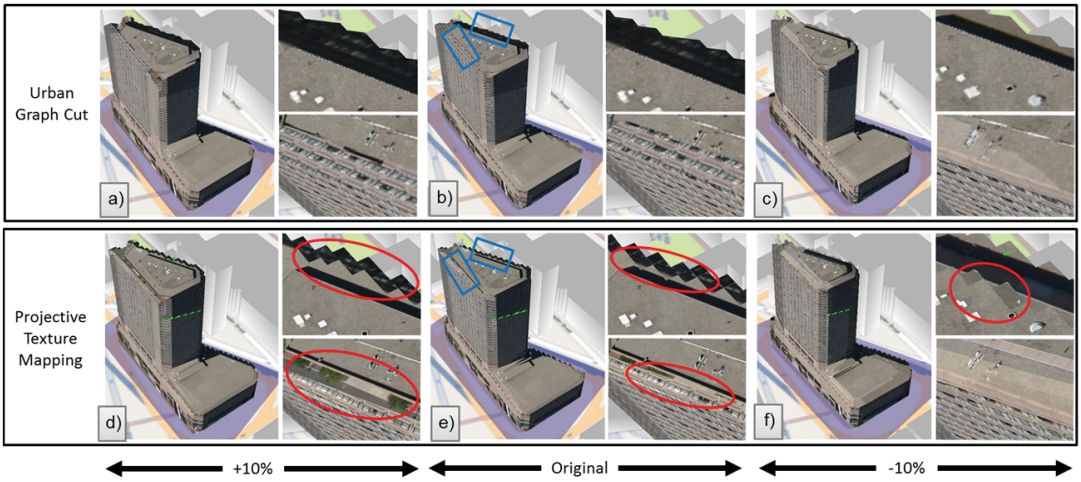

Fig. 5.29. **Graph-Cut vs. Projective Texture Mapping.** Comparison of our graph-cut algorithm with projective texture mapping for the original building and two altered proxies: building expanded +10% in all directions with random noise in the height map of 5m (left) and collapsed -10% in all directions with random noise in the height map of 5m (right). Our approach creates a seamless texture transition from facade to roof. In fact, as compared to projective texture mapping, it reduces the ill visual artifacts in all cases as can be seen by our results in the top row.

right picture of 14f). However, the smoothness of the image transition is never worse than the original.

Figure 5.30 contains a comparison of building-ground surface graph-cuts. For the building in Figure 5.30a, Figures 5.30b and 5.30c show the result using our proxies and standard projective texture mapping. By enabling the computation of building-ground surface graph-cuts, we are able to improve the coherence at the interface of the building and ground surfaces, as is seen in Figures 5.30d and 5.30e. In particular, notice the discontinuity of the roads and cars in Figures 5.30b and the projection of the extra roof surface in Figure 5.30c both of which are eliminated in our result.

Figure 5.31 contains an example of the benefit of our ground surface graph-cuts. Figure 4.1a contains the initial top-down view of an example area (we choose a camera with a view direction that is closest to the vertical axis). Observe how the building in

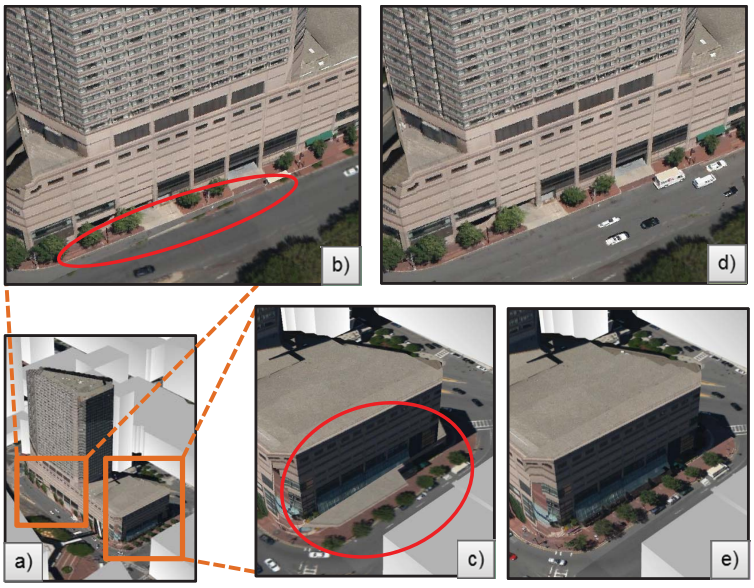

Fig. 5.30. **Building-Ground Surface Graph-Cuts.** a) We show two close-ups of this building. b-c) With projective texture mapping, there are discontinuities, missing content, and building projections at the boundary between the building and the street. d-e) Our building-ground surface graph-cuts are able to find a smooth transition between the two structures and produce a coherent and visually plausible appearance.

the middle occludes some of the nearby roads and buildings. Figure 5.31b contains the result of a nave graph-cut without taking into account the buildings proxies notice the disturbing visual artifacts despite the attempt of minimizing neighboring pixel differences with the graph-cut. Figure 5.31c shows the result of our ground surface graph-cut: buildings are not rendered on purpose and the occluded road pixels are automatically filled-in using content from other images. Figure 5.31d contains an image of the ground surface from Google Earth. Figure 5.31e shows the visual quality of our method using proxies and the ground surface from c. In contrast, using Googles ground images (Figure 5.31f) yields similar disturbing artifacts as in b.

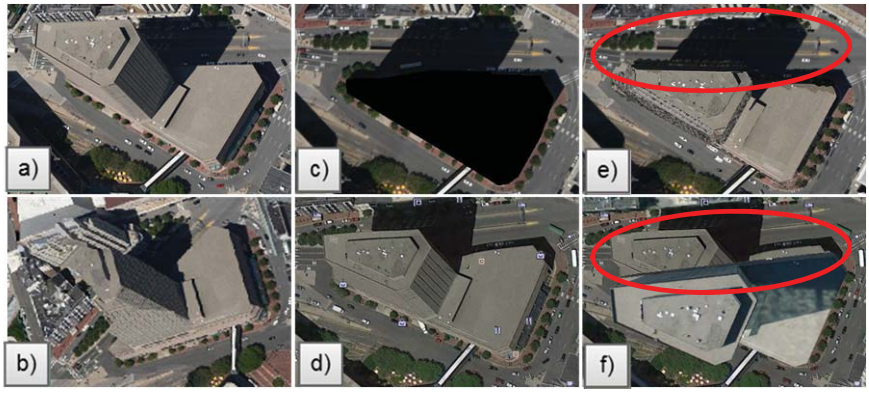

Fig. 5.31. **Ground Surface Graph-Cuts.** a) A downward looking original aerial image in our dataset (note occluded roads). b) Visual artifacts of using a nave graph-cut due to ignored inter-building occlusions. c) The result when using our ground surface graph-cut method. d) An image of the ground surface from Google Earth with no building proxies. e) Our method using building proxies and the ground from c. f) Using Google Earth imagery in projective texture mapping with buildings yields similar bad artifacts as in b.

Fig. 5.32. **Full Dataset View.** We show a birds eye view of the textured 3D model produced by our system.

Urban-Scale Reconstruction

We show in Figures 4.1, 5.32, and 5.33, several birds eye views of urban-scale examples (i.e., a fragment or portion of a city). Figures 4.1 and 5.32 show views of

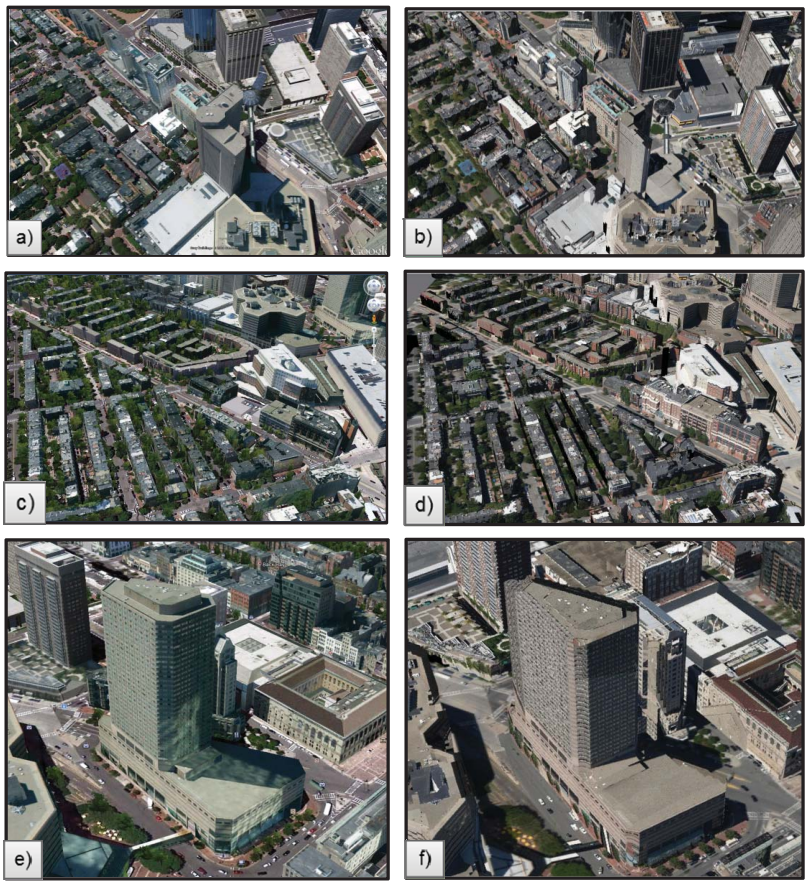

Fig. 5.33. **Google Earth Comparisons.** We show several comparisons between Google Earth snapshots (a,c,e) and our result (b,d,f). Our method yields similar quality results in most cases and thus opens up the door for the rapid creation of city-scale 3D models.

Boston reconstructed using our method. Figure 5.33 shows some close-ups of several city areas and the views using Google Earth, including its crowd-sourced buildings. It is important to note that Google Earth is using a *different image set* than ours though qualitatively similar and its models are all manually created. Our method is able to automatically produce good geometric proxies and to use surface graph-cuts to stitch together the aerial imagery yielding visually effective texture mapping.

5.4.2 Planer-Hinge Reconstruction

We use images in five viewing directions of resolution 5616 x 3744 in order to reconstruct 20 buildings. Table 5.2 shows the average computation time for one building:

Table 5.2.
Reconstruction Time. Time to compute each step of our method.

Bundler	**PMVS**	**Hinge**	**Plane**	**Recons.**
18 min	20 min	5 min	2 min	1 min

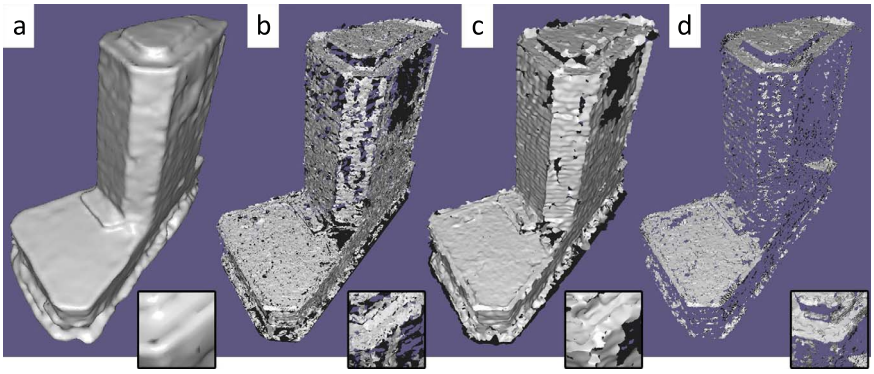

Fig. 5.34. **Triangulation Comparison.** a) Poisson reconstruction b) Grid Projection reconstruction c) RIMLS reconstruction d) Greedy projection triangulation.

Figure 5.34 shows the model reconstruction using one of four different triangulation algorithms. Poisson reconstruction generates a watertight closed model of the cloud point (4a). Marching cubes RIMLS reconstruction [100] is qualitatively similar to the one generated by Poisson reconstruction. However, it is not guaranteed to be closed and has many discontinuities (4c). Grid projection reconstruction [152] is not

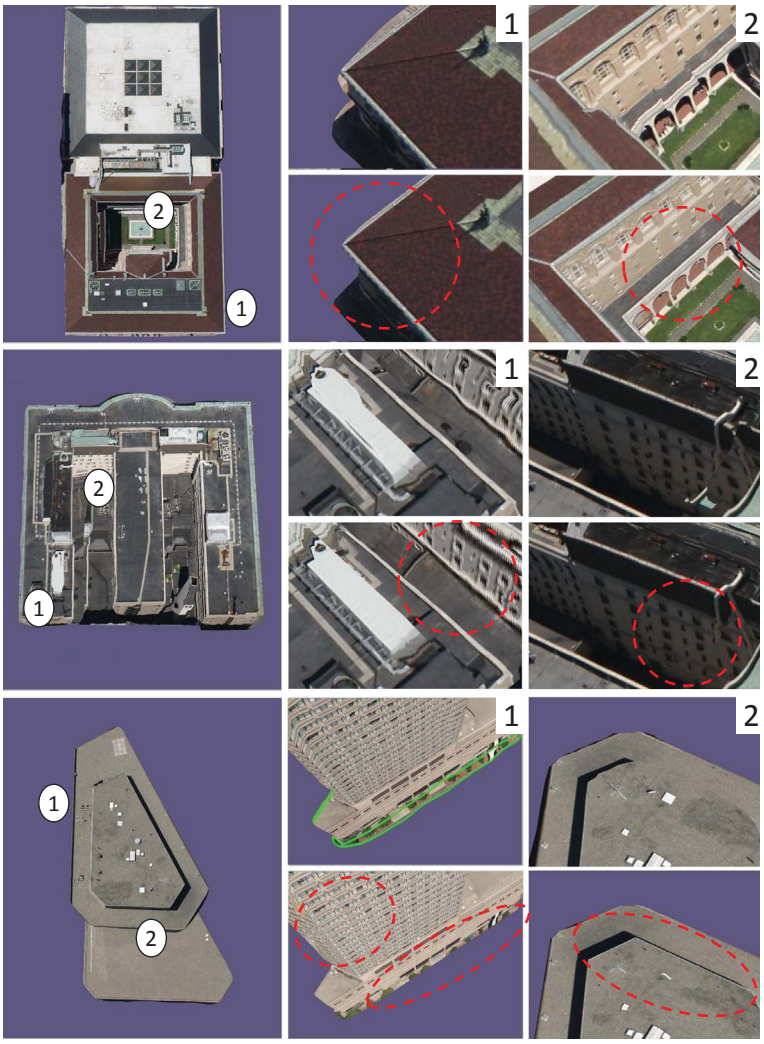

Fig. 5.35. **Our Results vs. Poisson Reconstruction.** Each row presents a comparison: top Poisson reconstruction, bottom our results.

able to fill holes (4b). Greedy projection triangulation [153] produces reconstructions quickly but is very sensitive to noise and holes (4d).

Figure 5.35 shows the improvement of our system as compared to a naïve Poisson reconstruction. Our system recovers sharper edges and corners using the hinge and plane constraints. In particular, the plane constraint significantly eliminates aberra-

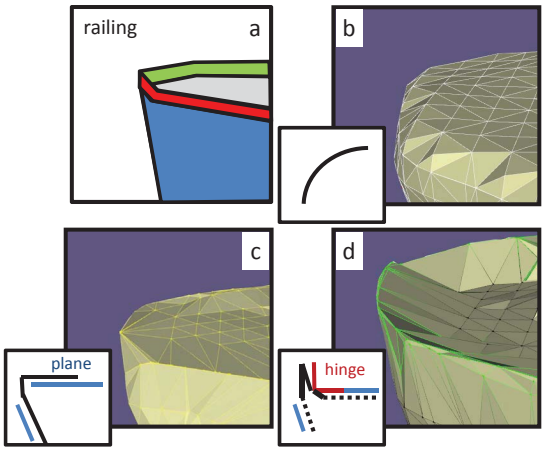

Fig. 5.36. **Results after using planes and hinges.** a) Actual geometry b) Poisson reconstruction c) After our plane reconstruction d) After our plane and hinge reconstruction.

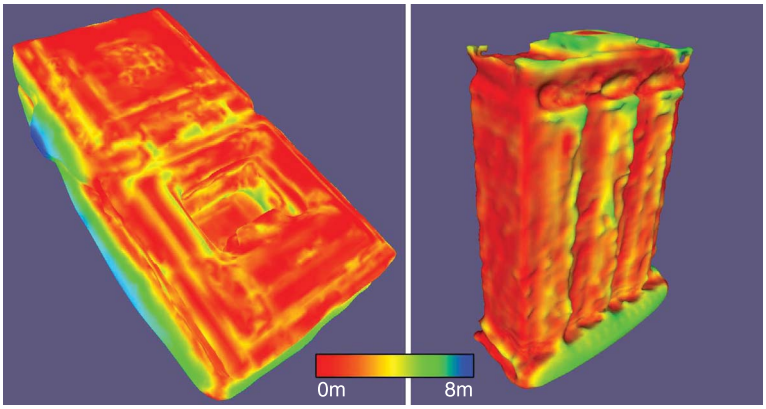

Fig. 5.37. **Surface displacement caused by our method.** Hausdorff Distance between Poisson surface and our final mode.

tions within facades. The hinge constraint improves the sharpness of the edges and corners (top) and can make incorrect geometry disappear (bottom).

Figure 5.36 shows the successive improvements of hinge and plane constraints. The initial model reconstruction has dull edges and perturbations in the supposedly flat parts due to lack of points and presence of noise (6b). Plane logic flattens the area,

making the building more rectilinear (6c). Hinge logic brings improved geometric details to the building and creates sharper edges where detected (6d).

Figure 5.37 shows for two building examples the Hausdorff Distance between Poisson surface and our final model. In the already flat areas, the improvement is small (red), but in the areas where the error is big (e.g., missing samples), our system improves the model significantly with surface displacements of up to 8 meters.

5.5 Summary

In this chapter, we have presented the results of this dissertation. We first explored our forward and inverse design for urban models, then traffic, and weather simulators. Finally, we concluded with the urban reconstruction.

6. CONCLUSIONS AND FUTURE WORK

We have presented a framework to enhance urban procedural modeling with inverse design controls. Our system allows the user control and design geometry, traffic, and weather through a set of high-level indicators or the direct control of the input parameter values.

Urban Procedural Modeling

We have coupled an automatic inverse design approach for urban procedural modeling with forward procedural modeling. Urban indicators are intuitive metrics for measuring the desirability of urban areas, and we have incorporated this as a key method for designers to efficiently generate optimized 3D urban models that meet their target criteria. The relationship of indicators to the procedural model is in general unknown and complex that has until now hindered their direct specification. We tackle the well-known open problem of controlling procedural modeling by providing a generalized mechanism that allows users to specify arbitrary target indicators and automatically compute the optimal parameters to obtain the desired output. Our methodology uses MCMC and artificial neural networks, including algorithms to search both local and global state changes, and presents multiple distinct 3D model options to the user.

Traffic Modeling

We have developed a super fast novel traffic microsimulation that runs on real-world road networks (OpenStreetMaps) and procedural models. The high performance of our simulator allows us create an inverse tool to design and control traffic. Using our framework, the user can interactively "painting" a desired vehicular traffic behavior (i.e., animation) and let the system to automatically computes a realistic 3D

urban model that yields the user-specified behavior. Our traffic manipulation strategy adapts an MCMC method to explore the solution space by performing a set of topology-preserving and topology-changing road network changes. We used our system to control traffic behaviors such as road occupancy, travel time, and CO emission.

Weather Modeling

We have also developed a super real-time rates (up to 4800 faster than real-time) realistic, physically-based weather model. Our weather simulation is based on a non-hydrostatic weather model consisting of a set of nonlinear dynamical equations which govern atmospheric motions. Our system allows the user to control and design weather. As validation, we compare our system against the well-known state-of-the-art weather forecast results and systems. This model also uses an MCMC method to explore the solution space, allowing different optimization variations depending on the purpose. We use our system to control the cloud coverage, rain, temperature of a 3D procedural model.

Reconstruction

We have presented two automatic urban-scale modeling approach using planar-hinge model, volumetric reconstruction from aerial calibrated images, and surface graph-cut based texture generation.

6.1 Future work

For our current framework, we have identified several limitations and future work items:

- **Urban Procedural Modeling** We would be interesting to explore alternative means to support scaling to a much larger number of parameters while keeping the accuracy. Moreover, we will explore additional indicators, including feeding

indicator values back to the model so as to, for instance, alter window sizes and wall materials based on the result of sun light exposure.

- **Traffic Modeling** We would be interested in exploring additional topology-preserving changes, such as altering intersection type and speed limit. Additionally, our simulator will be improved to support more complex traffic lights, on/off ramps, random driver behavior, and more.

- **Weather Modeling** First, since our focus is urban-scale our model cannot simulate weather phenomena that are formed on bigger or smaller scale; we will explore a multi-resolution grids to address this and enhance the global design. Second, our microphysics model currently simulates cumulus clouds and rain. We will explore more complex models to add snow and hail. Third, we will include additional land use categories including modeling the effect of terrain height on the land use properties and weather grid variables. Fourth, we will explore the use of shared memory, dynamic parallelism, and streaming to enhance GPU performance. Fifth, we will explore physically based weathering of urban models, such as buildings, using our weather simulator.

- **Reconstruction** We will incorporate knowledge of roads, sidewalks, and other urban structures, merge other data sources (e.g., LIDAR), and experiment with faster GPU implementations. Moreover, we would be interested in closing the loop between graph-cut calculation and proxy computation; e.g., an iterative process going between refinement of the proxy and re-computing graph-cuts.

REFERENCES

REFERENCES

[1] D. Manocha and M. C. Lin, "Interactive large-scale crowd simulation," in *Digital Urban Modeling and Simulation*. Springer, 2012, pp. 221–235.

[2] J. Sewall, D. Wilkie, and M. C. Lin, "Interactive hybrid simulation of large-scale traffic," in *ACM Transactions on Graphics*, vol. 30, no. 6. ACM, 2011, p. 135.

[3] Y. Gotanda, M. Kawase, and M. Kakimoto, "Real-time rendering of physically based optical effects in theory and practice," in *ACM SIGGRAPH 2015 Courses*. ACM, 2015, p. 23.

[4] R. Ando, N. Thuerey, and C. Wojtan, "A stream function solver for liquid simulations," *ACM Transactions on Graphics*, vol. 34, no. 4, p. 53, 2015.

[5] M. Macklin and M. Müller, "Position based fluids," *ACM Transactions on Graphics*, vol. 32, no. 4, p. 104, 2013.

[6] A. Lindenmayer, "Mathematical models for cellular interactions in development i. filaments with one-sided inputs," *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 280–299, 1968.

[7] S. Pirk, T. Niese, O. Deussen, and B. Neubert, "Capturing and animating the morphogenesis of polygonal tree models," *ACM Transactions on Graphics*, vol. 31, no. 6, pp. 169:1–169:10, Nov. 2012.

[8] J.-D. Génevaux, E. Galin, E. Guérin, A. Peytavie, and B. Beneš, "Terrain generation using procedural models based on hydrology," *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 143:1–143:13, Jul. 2013.

[9] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Computer Graphics and Interactive Techniques*. ACM, 2001, pp. 301–308.

[10] M. Honda, K. Mizuno, Y. Fukui, and S. Nishihara, "Generating autonomous time-varying virtual cities," in *International Conference on Cyberworlds*. IEEE Computer Society, 2004, pp. 45–52.

[11] B. Weber, P. Müller, P. Wonka, and M. Gross, "Interactive Geometric Simulation of 4D Cities," *Computer Graphics Forum*, vol. 28, no. 2, pp. 481–492, 2009.

[12] C. A. Vanegas, D. G. Aliaga, B. Beneš, and P. A. Waddell, "Interactive design of urban spaces using geometrical and behavioral modeling," *ACM Transactions on Graphics*, vol. 28, no. 5, pp. 111:1–111:10, Dec. 2009.

[13] O. Šťava, B. Beneš, R. Měch, D. G. Aliaga, and P. Krištof, "Inverse Procedural Modeling by Automatic Generation of L-systems," *Computer Graphics Forum*, vol. 29, no. 2, pp. 665–674, 2010.

[14] M. Bokeloh, M. Wand, and H.-P. Seidel, "A connection between partial symmetry and inverse procedural modeling," *ACM Transactions on Graphics*, vol. 29, pp. 104:1–104:10, July 2010.

[15] D. G. Aliaga, P. A. Rosen, and D. R. Bekins, "Style grammars for interactive visualization of architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 4, pp. 786–797, Jul. 2007.

[16] C. A. Vanegas, D. G. Aliaga, and B. Beneš, "Building reconstruction using manhattan-world grammars," *IEEE CVPR*, pp. 358–365, 2010.

[17] P. Müller, G. Zeng, P. Wonka, and L. Van Gool, "Image-based procedural modeling of facades," *ACM Transactions on Graphics*, vol. 26, no. 3, July 2007.

[18] J. Xiao, T. Fang, P. Tan, P. Zhao, E. Ofek, and L. Quan, "Image-based facade modeling," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 161:1–161:10, Dec. 2008.

[19] M. Bertero, T. Poggio, and V. Torre, "Ill-posed problems in early vision," *Proceedings of the IEEE*, vol. 76, no. 8, pp. 869 –889, Aug. 1988.

[20] T. P. Murphy, *Urban indicators: A guide to information sources*. Gale Research Co., 1980.

[21] C. Wong, *Indicators for Urban and Regional Planning: The Interplay of Policy and Methods*. Routledge, 2006.

[22] R. A. Pielke Sr, *Mesoscale meteorological modeling*. Academic press, 2013, vol. 98.

[23] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 669–677, Jul. 2003.

[24] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 614–623, Jul. 2006.

[25] CityEngine, "http://www.esri.com/software/cityengine."

[26] S. P. Arya, *Air pollution meteorology and dispersion*. Oxford University Press New York, 1999.

[27] TRANSECT, *Center for Applied Transect Studies*. www.transect.org, 2012.

[28] CNU, "Congress for the new urbanism," www.cnu.org, 2012.

[29] CTOD, "Center for transit-oriented development," www.ctod.org, 2012.

[30] M. Lipp, D. Scherzer, P. Wonka, and M. Wimmer, "Interactive modeling of city layouts using layers of procedural content," *Computer Graphics Forum*, vol. 30, no. 2, pp. 345–354, 2011.

[31] J. Go, T. D. Vu, and J. J. Kuffner, "Autonomous behaviors for interactive vehicle animations," *Graphical Models*, vol. 68, no. 2, pp. 90–112, 2006.

[32] J. Sewall, J. van den Berg, M. C. Lin, and D. Manocha, "Virtualized traffic: Reconstructing traffic flows from discrete spatiotemporal data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 1, pp. 26–37, 2011.

[33] J. Sewall, D. Wilkie, P. Merrell, and M. C. Lin, "Continuum traffic simulation." *Computer Graphics Forum*, vol. 29, no. 2, pp. 439–448, 2010.

[34] D. Wilkie, J. Sewall, and M. Lin, "Flow reconstruction for data-driven traffic animation," *ACM Transactions on Graphics*, vol. 32, no. 4, pp. 89–98, 2013.

[35] C. A. Vanegas, I. Garcia-Dorado, D. G. Aliaga, B. Benes, and P. Waddell, "Inverse design of urban procedural models," *ACM Transactions on Graphics*, vol. 31, no. 6, p. 168, 2012.

[36] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, "Interactive procedural street modeling," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 103–112, 2008.

[37] E. Galin, A. Peytavie, E. Gurin, and B. Benes, "Authoring Hierarchical Road Networks," *Computer Graphics Forum*, vol. 30, no. 7, pp. 2021–2030, 2011.

[38] "Multi-Agent Transactions Sim." {matsim.org}.

[39] Q. Yang and H. N. Koutsopoulos, "A microscopic traffic simulator for evaluation of dynamic traffic management systems," *Transactions Research Part C*, vol. 4, no. 3, pp. 113–129, 1996.

[40] "Simulation of Urban MObility," {sumo-sim.org}.

[41] "PTV VISSIM," {vision-traffic.ptvgroup.com}.

[42] J. Lebacque, S. Mammar, and H. Salem, "Generic second order traffic flow modeling," *Transactions and Traffic Theory*, pp. 755–776, 2007.

[43] D. Ngoduy, "Multiclass first-order traffic model using stochastic fundamental diagrams," *Transportmetrica*, vol. 7, no. 2, pp. 111–125, 2011.

[44] H. J. Payne, *Models of freeway traffic and control.* Simulation Councils, 1971.

[45] M. Treiber, A. Hennecke, and D. Helbing, "Congested traffic states in empirical observations and microscopic simulations," *Phys. Rev. E*, vol. 62, pp. 1805–1824, 2000.

[46] C. Choudhury, T. Toledo, and M. Ben-Akiva, "Modeling cooperative lane changing and forced merging behavior," *Transactions Research Board*, 2007.

[47] Y. Dobashi, K. Kusumoto, T. Nishita, and T. Yamamoto, "Feedback control of cumuliform cloud formation based on computational fluid dynamics," *ACM Transactions on Graphics*, vol. 27, no. 3, p. 94, 2008.

[48] C. Yuan, X. Liang, S. Hao, Y. Qi, and Q. Zhao, "Modelling cumulus cloud shape from a single image," in *Computer Graphics Forum*, vol. 33, no. 6. Wiley Online Library, 2014, pp. 288–297.

[49] M. J. Harris and A. Lastra, "Real-time cloud rendering," in *Computer Graphics Forum*, vol. 20, no. 3. Wiley Online Library, 2001, pp. 76–85.

[50] J. T. Kajiya and B. P. Von Herzen, "Ray tracing volume densities," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 165–174.

[51] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita, "A simple, efficient method for realistic animation of clouds," in *Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 19–28.

[52] R. Miyazaki, S. Yoshida, Y. Dobashi, and T. Nishita, "A method for modeling clouds based on atmospheric fluid dynamics," in *Computer Graphics and Applications*. IEEE, 2001, pp. 363–372.

[53] D. Overby, Z. Melek, and J. Keyser, "Interactive physically-based cloud simulation," in *Computer Graphics and Applications*. IEEE, 2002, pp. 469–470.

[54] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra, "Simulation of cloud dynamics on graphics hardware," in *Proceedings of Graphics Hardware*. Eurographics Association, 2003, pp. 92–101.

[55] J. R. Holton and G. J. Hakim, *An introduction to dynamic meteorology*. Academic press, 2012, vol. 88.

[56] Y. Zheng, K. V. Alapaty, J. Herwehe, A. Del Genio, and D. S. Niyogi, "Improving High-resolution Weather Forecasts using the Weather Research and Forecasting (WRF) Model with Upgraded Kain-Fritsch cumulus Scheme," *Monthly Weather Review*, May 2015.

[57] F. Nebeker, *Calculating the weather: Meteorology in the 20th century*. Academic Press, 1995, vol. 60.

[58] S. Roland, "Meteorology for scientists and engineers," *Brooks/Cole*, 2000.

[59] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, "A description of the advanced research wrf version 2," DTIC Document, Tech. Rep., 2005.

[60] F. Chen, H. Kusaka, R. Bornstein, J. Ching, C. Grimmond, S. Grossman-Clarke, T. Loridan, K. W. Manning, A. Martilli, S. Miao *et al.*, "The integrated wrf/urban modelling system: Development, evaluation, and applications to urban environmental problems," *International Journal of Climatology*, vol. 31, no. 2, pp. 273–288, 2011.

[61] W. R. Cotton, R. Pielke Sr, R. Walko, G. Liston, C. Tremback, H. Jiang, R. McAnelly, J. Harrington, M. Nicholls, G. Carrio *et al.*, "Rams 2001: Current status and future directions," *Meteorology and Atmospheric Physics*, vol. 82, no. 1-4, pp. 5–29, 2003.

[62] D. R. Durran, *Numerical Methods for Wave Equations in Geophysical Fluid Dynamics*. Springer Science & Business Media, 2013, vol. 32.

[63] A. Arakawa and V. R. Lamb, "Computational design of the basic dynamical processes of the ucla general circulation model," *Methods in Computational Physics*, vol. 17, pp. 173–265, 1977.

[64] R. Purser and L. Leslie, "A semi-implicit, semi-lagrangian finite-difference scheme using hligh-order spatial differencing on a nonstaggered grid," *Monthly Weather Review*, vol. 116, no. 10, pp. 2069–2080, 1988.

[65] F. Mesinger, G. DiMego, E. Kalnay, K. Mitchell, P. C. Shafran, W. Ebisuzaki, D. Jovic, J. Woollen, E. Rogers, E. H. Berbery *et al.*, "North American regional reanalysis," *Bulletin of the American Meteorological Society*, vol. 87, no. 3, pp. 343–360, 2006.

[66] E. Kessler, *On the distribution and continuity of water substance in atmospheric circulation*. American Meteorological Society, 1969.

[67] S.-T. Soong and Y. Ogura, "A comparison between axisymmetric and slab-symmetric cumulus cloud models," *Journal of the Atmospheric Sciences*, vol. 30, no. 5, pp. 879–893, 1973.

[68] O. Tetens, "Über einige meteorologische Begriffe," *Zeitschrift für Geophysik*, vol. 6, pp. 297–309, 1930.

[69] R. B. Stull, *An introduction to boundary layer meteorology*. Springer Science & Business Media, 1988, vol. 13.

[70] W. D. Solecki, C. Rosenzweig, L. Parshall, G. Pope, M. Clark, J. Cox, and M. Wiencke, "Mitigation of the heat island effect in urban new jersey," *Global Environmental Change Part B: Environmental Hazards*, vol. 6, no. 1, pp. 39–49, 2005.

[71] A. Blackadar, "Modeling pollutant transfer during daytime convection," in *Symposium on Turbulence, Diffusion, and Air Pollution, 4 th, Reno, Nev*, 1978, pp. 443–447.

[72] F. Chen and J. Dudhia, "Coupling an advanced land surface-hydrology model with the penn state-ncar mm5 modeling system. part i: Model implementation and sensitivity," *Monthly Weather Review*, vol. 129, no. 4, pp. 569–585, 2001.

[73] B. Beneš, O. Šťava, R. Měch, and G. Miller, "Guided Procedural Modeling," *Computer Graphics Forum*, pp. 325–334, 2011.

[74] J. P. Park, K. H. Lee, and J. Lee, "Finding syntactic structures from human motion data," *Computer Graphics Forum*, vol. 30, no. 8, pp. 2183–2193, 2011.

[75] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, and V. Koltun, "Interactive furniture layout using interior design guidelines," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 87:1–87:10, Jul. 2011.

[76] L.-F. Yu, S.-K. Yeung, C.-K. Tang, D. Terzopoulos, T. F. Chan, and S. J. Osher, "Make it home: automatic optimization of furniture arrangement," *ACM Transactions on Graphics*, vol. 30, no. 4, pp. 86:1–86:12, July 2011.

[77] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun, "Metropolis procedural modeling," *ACM Transactions on Graphics*, vol. 30, no. 2, p. 11, 2011.

[78] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, and B. Benes, "Inverse procedural modelling of trees," *Computer Graphics Forum*, vol. 33, no. 6, pp. 118–131, 2014.

[79] D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan, "Controlling procedural modeling programs with stochastically-ordered sequential monte carlo," *ACM Transactions on Graphics*, vol. 34, no. 4, pp. 105:1–105:11, Jul. 2015.

[80] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[81] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[82] M. Batty, *Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals*. MIT Press, 2007.

[83] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," *Ph.D, dissertation, Harvard Univ.*, 1974.

[84] M. L. Minsky and S. A. Papert, *Perceptrons-Expanded Edition: An Introduction to Computational Geometry*. MIT Press Boston, MA, 1987.

[85] S. Sharma, S. Ukkusuri, and T. Mathew, "Pareto optimal multiobjective optimization for robust transportation network design problem," *Transactions Research Record: Journal of the Transactions Research Board*, vol. 2090, pp. 95–104, 2009.

[86] H. v. Stackelberg, D. Bazin, L. Urch, and R. R. Hill, *Market Structure and Equilibrium*. Springer, 2011.

[87] P. Waddell, "Urbansim: Modeling urban development for land use, transportation and environmental planning," *Journal of American Planning Association*, vol. 68, pp. 297–314, 2002.

[88] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast accurate computation of large-scale ip traffic matrices from link loads," *Proceedings of Sigmetrics*, pp. 206–217, 2003.

[89] H. M. A. Aziz and S. V. Ukkusuri, "Integration of environmental objectives in a system optimal dynamic traffic assignment model," *Computer-Aided Civil and Infrastructure Engineering*, vol. 27, no. 7, pp. 494–511, 2012.

[90] P. Musialski, P. Wonka, D. G. Aliaga, M. Wimmer, L. van Gool, and W. Purgathofer, "A survey of urban reconstruction," *Computer Graphics Forum*, vol. 32, no. 6, pp. 146–177, 2013.

[91] J. Xiao, T. Fang, P. Zhao, M. Lhuillier, and L. Quan, "Image-based street-side city modeling," *ACM Transactions on Graphics*, vol. 28, no. 5, pp. 114:1–114:12, Dec. 2009.

[92] O. Teboul, I. Kokkinos, L. Simon, P. Koutsourakis, and N. Paragios, "Shape grammar parsing via reinforcement learning," in *IEEE CVPR*. IEEE Computer Society, 2011, pp. 2273–2280.

[93] F. Lafarge, R. Keriven, M. Bredif, and V. H. Hiep, "Hybrid multi-view reconstruction by jump-diffusion," in *IEEE CVPR*, June 2010, pp. 350–357.

[94] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski, "Reconstructing building interiors from images," in *IEEE ICCV*, 2009.

[95] H.-H. Liao, Y. Lin, and G. Medioni, "Aerial 3D reconstruction with line-constrained dynamic programming," in *ICCV*. IEEE Computer Society, 2011, pp. 1855–1862.

[96] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. Seitz, "Multi-view stereo for community photo collections," in *ICCV*, Oct 2007, pp. 1–8.

[97] S. Agarwal, N. Snavely, I. Simon, S. Seitz, and R. Szeliski, "Building rome in a day," in *Computer Vision*, Sept 2009, pp. 72–79.

[98] J.-M. Frahm, P. Fite-Georgel, D. Gallup, T. Johnson, R. Raguram, C. Wu, Y.-H. Jen, E. Dunn, B. Clipp, S. Lazebnik, and M. Pollefeys, "Building rome on a cloudless day," in *ECCV*, 2010, pp. 368–381.

[99] J. Daniels II, T. Ochotta, L. K. Ha, and C. T. Silva, "Spline-based feature curves from point-sampled geometry," *Visual Computing*, 2008.

[100] C. Öztireli, G. Guennebaud, and M. Gross, "Feature preserving point set surfaces based on non-linear kernel regression," in *Eurographics*, 2009.

[101] A.-L. Chauve, P. Labatut, and J.-P. Pons, "Robust piecewise-planar 3d reconstruction and completion from large-scale unstructured point data," in *IEEE CVPR*, june 2010.

[102] L. Nan, A. Sharf, H. Zhang, D. Cohen-Or, and B. Chen, "Smartboxes for interactive urban reconstruction," *ACM Transactions on Graphics*, vol. 29, no. 4, pp. 93:1–93:10, Jul. 2010.

[103] Q. Zheng, A. Sharf, G. Wan, Y. Li, N. J. Mitra, D. Cohen-Or, and B. Chen, "Non-local scan consolidation for 3d urban scenes," *ACM Transactions on Graphics*, vol. 29, no. 4, pp. 94:1–94:9, Jul. 2010.

[104] Q.-Y. Zhou and U. Neumann, "2.5D dual contouring: A robust approach to creating building models from aerial LiDAR point clouds," in *ECCV*. Springer-Verlag, 2010, pp. 115–128.

[105] C. Poullis and S. You, "Photorealistic large-scale urban city model reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 4, pp. 654–669, Jul. 2009.

[106] F. Lafarge and C. Mallet, "Building large urban environments from unstructured point data," in *Proceedings of ICCV*. IEEE Computer Society, 2011, pp. 1068–1075.

[107] C.-H. Shen, S.-S. Huang, H. Fu, and S.-M. Hu, "Adaptive partitioning of urban facades," *ACM Transactions on Graphics*, vol. 30, no. 6, pp. 184:1–184:10, Dec. 2011.

[108] A. Toshev, P. Mordohai, and B. Taskar, "Detecting and parsing architecture at city scale from range data," in *IEEE CVPR*, June 2010, pp. 398–405.

[109] A. Golovinskiy, V. Kim, and T. Funkhouser, "Shape-based recognition of 3d point clouds in urban environments," in *Computer Vision*, Sept 2009, pp. 2154–2161.

[110] M. Ding, K. Lyngbaek, and A. Zakhor, "Automatic registration of aerial imagery with untextured 3D LiDAR models," in *IEEE CVPR*, June 2008, pp. 1–8.

[111] L. Wang and U. Neumann, "A robust approach for automatic registration of aerial images with untextured aerial LiDAR data," in *IEEE CVPR*, June 2009, pp. 2623–2630.

[112] C. Frueh, R. Sammon, and A. Zakhor, "Automated texture mapping of 3d city models with oblique aerial imagery," in *3DPVT*, Sept 2004, pp. 396–403.

[113] K. Kutulakos and S. Seitz, "A theory of shape by space carving," in *Computer Vision*, vol. 1, 1999, pp. 307–314 vol.1.

[114] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan, "Image-based visual hulls," in *Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 369–374.

[115] D. Gallup, M. Pollefeys, and J.-M. Frahm, "3d reconstruction using an n-layer heightmap," in *Proceedings of DAGM*. Springer-Verlag, 2010, pp. 1–10.

[116] T. Pollard and J. Mundy, "Change detection in a 3-d world," in *IEEE CVPR*, June 2007, pp. 1–6.

[117] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: Image and video synthesis using graph cuts," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 277–286, Jul. 2003.

[118] S. Lefebvre, S. Hornus, and A. Lasram, "By-example synthesis of architectural textures," *ACM Transactions on Graphics*, vol. 29, no. 4, pp. 84:1–84:8, Jul. 2010.

[119] G. Vogiatzis, P. H. S. Torr, and R. Cipolla, "Multi-view stereo via volumetric graph-cuts," in *IEEE CVPR*. IEEE Computer Society, 2005, pp. 391–398.

[120] S. N. Sinha and M. Pollefeys, "Multi-view reconstruction using photo-consistency and exact silhouette constraints: A maximum-flow formulation," in *Proceedings of ICCV*. IEEE Computer Society, 2005, pp. 349–356.

[121] S. Tran and L. Davis, "3d surface reconstruction using graph cuts with surface constraints," in *ECCV*. Springer, 2006, pp. 219–231.

[122] V. Lempitsky and D. Ivanov, "Seamless mosaicing of image-based texture maps," in *IEEE CVPR*, June 2007, pp. 1–6.

[123] C. Allene, J.-P. Pons, and R. Keriven, "Seamless image-based texture atlases using multi-band blending," in *ICPR*, Dec 2008, pp. 1–4.

[124] Z. Gao, L. Nocera, and U. Neumann, "Fusing oblique imagery with augmented aerial LiDAR," in *SIGSPATIAL*. ACM, 2012, pp. 426–429.

[125] M. Mathias, A. Martinovic, J. Weissenberg, and L. V. Gool, "Procedural 3D Building Reconstruction Using Shape Grammars and Detectors," in *3DIMPVT*. IEEE Computer Society, 2011, pp. 304–311.

[126] F. Taillandier, "Automatic building reconstruction from cadastral maps and aerial images," in *Proceedings of the ISPRS Workshop CMRT 2005*, 2005.

[127] A. A. Montenegro, P. C. P. Carvalho, M. Gattass, and L. C. P. R. Velho, "Adaptive space carving," in *3DPVT*, 2004, pp. 199–206.

[128] S. Lazebnik, Y. Furukawa, and J. Ponce, "Projective visual hulls," *Int. J. Computer Vision*, vol. 74, no. 2, pp. 137–165, Aug. 2007.

[129] S. Shalom, A. Shamir, H. Zhang, and D. Cohen-Or, "Cone carving for surface reconstruction," *ACM Transactions on Graphics*, vol. 29, no. 6, pp. 150:1–150:10, Dec. 2010.

[130] M. H. F. Wilkinson and J. B. T. M. Roerdink, Eds., *ISMM*, vol. 5720. Springer, 2009.

[131] Y. Boykov and V. Kolmogorov, "Computing geodesics and minimal surfaces via graph cuts," in *Computer Vision*, Oct 2003, pp. 26–33 vol.1.

[132] T. K. Dey, X. Ge, Q. Que, I. Safa, L. Wang, and Y. Wang, "Feature-preserving reconstruction of singular surfaces," *Computer Graphics Forum*, 2012.

[133] J. Coughlan and A. Yuille, "Manhattan world: compass direction from a single image by bayesian inference," in *IEEE ICCV*, 1999.

[134] G. Schindler and F. Dellaert, "Atlanta world: an expectation maximization framework for simultaneous low-level edge grouping and camera calibration in complex man-made environments," in *IEEE CVPR*, 2004.

[135] N. Snavely, S. M. Seitz, and R. Szeliski, "Photo tourism: Exploring photo collections in 3D," in *ACM SIGGRAPH*, 2006.

[136] Y. Furukawa, B. Curless, S. Seitz, and R. Szeliski, "Towards internet-scale multi-view stereo," in *IEEE CVPR*, 2010.

[137] Y. Furukawa and J. Ponce, "Accurate, dense, and robust multi-view stereopsis," in *IEEE CVPR*, 2007.

[138] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Eurographics symposium on Geometry processing*, 2006.

[139] C. L. Zhang, F. Chen, S. G. Miao, Q. C. Li, X. A. Xia, and C. Y. Xuan, "Impacts of urban expansion and future green planting on summer precipitation in the beijing metropolitan area," *Journal of Geophysical Research: Atmospheres (1984–2012)*, vol. 114, no. D2, 2009.

[140] A. Gero, A. Pitman, G. Narisma, C. Jacobson, and R. Pielke, "The impact of land cover change on storms in the sydney basin, australia," *Global and Planetary Change*, vol. 54, no. 1, pp. 57–78, 2006.

[141] T. Lucke and P. W. Nichols, "The pollution removal and stormwater reduction performance of street-side bioretention basins after ten years in operation," *Science of The Total Environment*, vol. 536, pp. 784–792, 2015.

[142] J. Mullaney, T. Lucke, and S. J. Trueman, "A review of benefits and challenges in growing street trees in paved urban environments," *Landscape and Urban Planning*, vol. 134, pp. 157–166, 2015.

[143] WRP-URL, "www.whiteroofproject.org," the White Roof Project. Accessed: 2015-10.

[144] M. Santamouris, "Cooling the cities – a review of reflective and green roof mitigation technologies to fight heat island and improve comfort in urban environments," *Solar Energy*, vol. 103, pp. 682–703, 2014.

[145] J. Straka, R. B. Wilhelmson, L. J. Wicker, J. R. Anderson, and K. K. Droegemeier, "Numerical solutions of a non-linear density current: A benchmark solution and comparisons," *International Journal for Numerical Methods in Fluids*, vol. 17, no. 1, pp. 1–22, 1993.

[146] L. J. Wicker and W. C. Skamarock, "Time-splitting methods for elastic models using forward time schemes," *Monthly Weather Review*, vol. 130, no. 8, pp. 2088–2097, 2002.

[147] N. Ahmad and J. Lindeman, "Euler solutions using flux-based wave decomposition," *International Journal for Numerical Methods in Fluids*, vol. 54, no. 1, pp. 47–72, 2007.

[148] H. B. Bluestein and C. R. Parks, "A synoptic and photographic climatology of low-precipitation severe thunderstorms in the southern plains," *Monthly weather review*, vol. 111, no. 10, pp. 2034–2046, 1983.

[149] E. N. Lorenz, "Atmospheric predictability as revealed by naturally occurring analogues," *Journal of the Atmospheric sciences*, vol. 26, no. 4, pp. 636–646, 1969.

[150] D. Barker, X.-Y. Huang, Z. Liu, T. Auligné, X. Zhang, S. Rugg, R. Ajjaji, A. Bourgeois, J. Bray, Y. Chen *et al.*, "The weather research and forecasting model's community variational/ensemble data assimilation system: Wrfda," *Bulletin of the American Meteorological Society*, vol. 93, no. 6, pp. 831–843, 2012.

[151] F. Schmidt, E. Toppe, and D. Cremers, "Efficient planar graph cuts with applications in computer vision," in *IEEE CVPR*, June 2009, pp. 351–356.

[152] R. Li, L. Liu, L. Phan, S. Abeysinghe, C. Grimm, and T. Ju, "Polygonizing extremal surfaces with manifold guarantees," in *ACM Symp on Solid and Physical Modeling*, 2010.

[153] Z. C. Marton, R. B. Rusu, and M. Beetz, "On fast surface reconstruction methods for large and noisy point clouds," in *IEEE ICRA*, 2009.

VITA

VITA

Ignacio Garcia-Dorado received his Ph.D. in the Department of Computer Science at Purdue University. He worked as an research assistant under the supervision of Professor Daniel Aliaga. His doctoral research focused on inverse procedural modeling, 3D urban reconstruction, and human vision. Ignacio received an M.S. degree in electrical engineering from Universidad Politecnica de Madrid, Spain; an M.S. degree in computer engineering from Lunds Tekniska Högskola, Sweden, both in 2008; and an M.S. degree in computer science from Purdue University in 2014.

From 2008 to 2010, Ignacio worked at the European Space Agency as a computer engineer in Noordwijk, The Netherlands. From January to May 2010, he worked as a research assistant at McGill University in the Center of Intelligent Machines in Montreal, Canada. After this, he was awarded a Fulbright Grant to initiate Ph.D. studies at Purdue University, USA.

At Purdue, on the way to the Ph.D., he worked as a research intern for NVidia in summer 2013 and as an research assistant at U.C. Berkeley in summer 2014. After the completion of his Ph.D., Ignacio joined Google in Mountain View, CA, as a Ph.D. Software Engineer.