ARDA: A FRAMEWORK FOR PROCEDURAL VIDEO GAME CONTENT GENERATION

by

Nicholas Eugene Rudzicz

School of Computer Science
McGill University, Montreal

February 2009

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Abstract

The current trend in computer game design is toward larger and richer virtual worlds, providing interesting and abundant game content for players to explore. The creation and continuous expansion of detailed virtual environments, however, is a time and resource-consuming task for game developers. *Procedural content generation* potentially solves this problem; textures, landscapes, and more recently the creation of entire cities and their constituent roads and buildings can be performed in an automated fashion, potentially offering considerable resource savings for developers. This thesis develops a comprehensive system for content generation, centering on a hierarchical world design. The *Arda* tool is presented as a modular system for supporting automatic content generation in a game context. Arda is composed of a framework for internally managing environmental data and content-generation algorithms, in addition to a graphical tool providing a simplified interface for generating video game content. The tool supports extensive customization, which is demonstrated through a novel algorithm for efficient and realistic city (road-plan) generation. Experimental results show the design is practical, producing qualitatively realistic results. It is also quite fast; large city plans, extending to the size of major real-world cities can be generated in a few seconds. The Arda design demonstrates the feasibility of automatic content generation, providing a practical and flexible system for quick prototyping, development and testing of game assets.

Résumé

La nouvelle tendance dans la conception de jeux vidéo vise des environnements vastes et détaillés, pour mettre en valeur l'exploration de l'univers virtuel. Cependant, la création de ce contenu impose au développeur un investissement majeur au niveau des ressources et du temps requis pour la production. L'emploi du *contenu procédural* pourrait résoudre le problème : la création de textures, de paysages, et finalement de villes entières—incluant des réseaux routiers et des bâtiments complexes—se fait de façon automatisée, ce qui représente une économie considérable pour les développeurs. Cette thèse présente un système de génération de contenu procédural axé sur une conception hiérarchique du monde virtuel : *Arda*, un outil modulaire qui facilite la création automatique de contenu pour les jeux vidéo. Arda est composé d'un cadre d'applications pour la gestion de données environnementales et d'algorithmes procéduraux, ainsi que d'une interface d'utilisation graphique qui simplifie le processus de création de contenu. L'outil est fortement paramétrable, ce qui est démontré grâce à un nouvel algorithme, conçu ci-après, pour la création efficace et réaliste de réseaux routiers. Les essais prouvent que le concept est pratique et produit des résultats très réalistes. De plus, le système est rapide : des villes virtuelles, de la même dimension que des villes actuelles, sont construites en quelques secondes. Le système Arda démontre la faisabilité de la génération de contenu procédural, fournissant une méthode pratique et modulaire pour le dévelopement et la validation rapide de contenu pour les jeux vidéo.

Acknowledgments

I would like to acknowledge the help and supervision provided by Clark Verbrugge throughout the course of this project. It has been instrumental and inspiring.
A debt of gratitude, now and always, to my family. For the usual reasons.

"Ash nazg durbatulûk,
Ash nazg gimbatul,
Ash nazg thrakatulûk,
Agh burzum-ishi krimpatul!"

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Content in Video Games

Environmental content in video games has long played an important role in the gameplay experience. The earliest adventure games—both text-based and graphical; for example, the *Zork* or *King's Quest* series, respectively—rewarded exploration as a means of advancing the storyline. More recent "open world" games—such as the *Grand Theft Auto* series, *Far Cry 2*, *Fallout 3*, *The Elder Scrolls: Morrowind* and *Oblivion*, and any number of Massively Multiplayer Online (MMO) games—provide expansive environments generally tailored towards incrementally improving an avatar's characteristics. The latter type of environment—the open world—has in fact increased dramatically in popularity, as storage space has become more plentiful and less expensive on both the PC and home gaming consoles. At the same time, the graphics processing power of video game systems has also trended towards greater power for a lower price point; consequently, gamers expect much more detail from a game's environment. Taking the improvements in storage space and graphical power together, we note a trend in modern open world game environments: while this type of game world is becoming more common in games—and significantly more detailed—the actual *size* of such environments has been drastically reduced. For example, *The Elder Scrolls: Daggerfall*, a role-playing game released in 1996, boasted

"a game world the size of Great Britain", or roughly 209,000km^2 [2, 32]; in contrast, the latest game in the same series, *Oblivion*, occupies a mere 41km^2 [33].

Increased production costs are at the core of this dichotomy. Whereas *Daggerfall*'s wilderness environments were relatively sparse—a flat terrain arbitrarily populated by vegetation sprites—the world in *Oblivion* is much more detailed, with varying terrain, dense and realistic vegetation, even simulated wildlife (see Figure 1.1). Such high levels of realism and visual detail require a significant proportion of a game's development budget, in terms of both money spent on artists' salaries and necessary tool sets, and time required to complete the content and the game itself. Since the majority of video game companies still make use of a "brute force" approach to content creation—that is, simply hiring more artists and investing more time in the process—the costs of creating expansive environments, all while maintaining parity with continually-improving technology, will continue to spiral upwards, becoming prohibitively expensive for all but the largest companies with the greatest resources.

1.2 Procedural Content

An alternative to the brute-force approach is the use of *procedural content generation*, or PCG; that is, allowing content to be created and deployed by the computer program itself, by means of well-defined algorithms and/or rule sets. This enables vast amounts of content to be produced in a fraction of the cost and time normally required, potentially representing an important savings for companies creating their own open-world games. While this has granted PCG techniques a broad and rapidly growing interest in more recent years, it is worthwhile to note that a number of very early games have already used procedural content quite successfully, and furthermore that many of the most popular algorithms have been known for decades. In fact, research into PCG has proceeded quite steadily in a number of otherwise unrelated disciplines—simulation, plant biology, statistics, urban sociology, population studies,

(a)

(b)

Figure 1.1 – **Evolution of environmental content**. (a) An environment in *The Elder Scrolls: Daggerfall*; the distribution of trees and other details are procedural, if somewhat simple. (b) An environment from a more recent game in the same series, *Oblivion*. The environment is much more detailed, but relies primarily on manual content generation. *[Both games property of Bethesda Softworks.]*

and so on—and has resulted in a huge variety of well-studied algorithms (we discuss a number of these techniques, along with several games making use of PCG, in Chapter 2). However, every one of these techniques currently exists in isolation, producing solitary artefacts—visualisations, for the most part—that are interesting from a technical standpoint, and often very impressive aesthetically, but insufficient for a complete mainstream video game.

On the one hand, it is clear that (purely) manually-created environmental content will become increasingly expensive as time goes on; on the other, there exists a multitude of (currently isolated) techniques capable of generating a staggering variety of content entirely automatically, using a fraction of the resources of traditional asset creation. The increasing importance of the content-generation problem, coupled with the sophistication of procedural techniques as a potential solution, suggests that PCG will play a growing role in future game design, an observation that provides the main motivation for this thesis work.

1.3 Arda

First and foremost, our primary goal is to mediate the aforementioned discrepancy between the capability and variety of content generation algorithms, and their relative scarcity in modern video games—in other words, to make procedural content more accessible to game developers. To do so, we propose here a content generation system consisting of both a framework (essentially an application programming interface, or API), and a graphical toolkit (a graphical application to make creating content simple and intuitive). We refer to this system as Arda, and explain both the framework and the toolkit below. (By virtue of explanation, we note here that the name "Arda" was originally given by J.R.R. Tolkien to the vast and detailed world in which his *Silmarillion* and *Lord of the Rings* novels took place.)

There are a number of required and/or desirable features for a system such as Arda. In order to provide the most practical benefits to developers and researchers, we determined the following goals for the system:

- *Hierarchy*—Since most content creation algorithms target a specific type of content, Arda should maintain a clear distinction between these various types. Furthermore, there are a number of logical dependencies between types: buildings belong in a city, cities belong on a terrain, and so on. These dependencies should be reflected in the final structure of the system.

- *Modularity*—If two algorithms generate the same type of content—for instance, both Perlin noise and fractal methods can be used to generate terrain—then the system should allow either one to be "swapped" with the other in such a way that the replacement is transparent to the rest of the content creation process.

- *Simplicity*—To allow for the desired modularity, simple interfaces must be provided in various points of the framework, such that game developers and algorithm researchers can easily import their techniques into the system.

- *Parallelism*—Given that, in a single virtual world, there may be a large amount of non-overlapping content to generate, Arda should accommodate parallelised

computation, to improve performance and take advantage of modern multi-core systems.

- *Versatility*—Once created, content data should be stored in abstract, high-level internal data structures so that they can be exported later to a variety of platforms and/or formats.

- *Basic functionality*—The graphical tool for content creation should be able to generate a virtual world immediately upon starting, without the need for any up-front programming on the user's part. Therefore, several "default" algorithms should be created to allow full environment creation from start to finish.

These design principles guided the development of the Arda Framework and the Arda Toolkit presented in subsequent chapters.

1.4 A Hybrid Content Creation Approach

Finally, expanding on the goal of "Versatility" outlined above, we note that the Arda content-generating system is not meant to be monolithic in the process of asset creation. Rather, it is intended simply as one half of a "hybrid" content creation approach—a process whereby procedural and manually-created content can be rapidly combined to ensure both maximum efficiency and maximum artistry. In other words, while procedural techniques can lay content "ground-work" or create otherwise prohibitively-large environments automatically, human artists can still modify the resulting assets or introduce their own content in order to better suit a particular project's gameplay requirements or general aesthetic.

As previously stated, procedural content generation is not a recent innovation in the video games industry. Indeed, as early as 1985, the *Adventure Construction Set* (or ACS), developed by Electronic Arts, provided users with procedurally-generated content—of a much simpler nature than what is currently possible—and enabled a hybrid approach very similar to that which we propose with Arda: (*italics added*) "ACS also displayed a key truth reflected in later adventure building systems: they

only automated the mechanical parts of game construction; good game design was still difficult and time-consuming. The option to allow the computer to create a random adventure produced treasure-hunt mazes and compounds, but no story or logical succession of challenges to link them together. *Many users would use the random function to save time and create large, relatively empty canvases and then build their adventures by modifying them.*" [24, 30]

1.5 Contributions

- **Arda Framework**: An internal representation of the various types of content data to be created, with the ability to create large amounts of procedural content in a top-down approach. The Framework defines a number of points where human input would otherwise be required; these are replaced with "interfaces" which allow different algorithms to be swapped in and out of their place. Game developers can create new algorithms and very quickly adapt them to the system, extending the Framework's capabilities.

- **Arda Toolkit**: A graphical tool built on top of the Framework, providing quick and intuitive access to the latter's underlying functions. Users can generate extensive environments, select from different algorithms to perform content generation at any level, regenerate selected portions of the created content, and modify the parameters of any algorithm chosen. A simple preview of the resulting virtual worlds is provided; having evaluated the latter, the user is able to export the given world to a format of their choosing.

- **Iterated Subdivision Algorithm**: Part of the development of the Framework and Toolkit involved the implementation of content creation algorithms for evaluation purposes. The task of generating random, procedural urban road networks has been addressed elsewhere; however, previous techniques are complex and difficult to parametrise. We present, in conjunction with the development of Arda, a simpler, more intuitive road generation algorithm with results qualitatively approaching those of previous algorithms.

- **Experimental Trials**: Finally, we examine the performance of our road generation algorithm in terms of running time and aesthetic quality. Furthermore, we test the Arda content generation system by producing large environments for an extant video game platform, and evaluate the results.

In Chapter 2, we examine the various methods (both manual and procedural) used in generating content for video games; furthermore, we survey the state of the art in content-generation algorithms. Chapter 3 presents the Arda system for procedural content generation, including both an internal representation and a graphical tool. Next, Iterated Subdivision—a novel algorithm for urban road network generation—is developed in Chapter 4, with the goal of integrating it into the previously-defined Arda system. Finally, we evaluate the performance of both Iterated Subdivision and the Arda system itself in Chapter 5, before concluding in Chapter 6 and suggesting further possibilities for research.

Chapter 2
Related Work

As hinted in the previous chapter, procedural content generation (PCG) is not a recent development in the field of computer science in general, nor the video game industry in particular. Many of the earliest video games used procedural content in very important ways; meanwhile, a wide variety of disciplines within academia have continued to contribute inventive and powerful algorithms for the automatic creation of any number of different content types. In this chapter, we examine traditional methods of generating environmental content, as well as discussing previous uses of procedural content in video games, and the impressive range of procedural algorithms developed within the industry and within other research disciplines. Each of these concepts can inform the development of Arda, our own content generation system.

2.1 Existing Content Creation Techniques

Traditionally, video game asset creation has relied on general-purpose digital art software—for instance, many 2D games are developed using tools such as Adobe's *Illustrator* or *Photoshop Elements* to create sprites; whereas 3D games make use of any number of 3D modelling utilities, such as *Blender*, *3DS Max*, *Maya*, and so on. While these are powerful tools, they often require a significant amount of training in order to achieve practical results, and moreover, the content thus created still requires integration into a game environment, which can be a complicated task. For these

reasons, many game developers have designed tool sets to easily manage individual content elements created with the utilities listed above, and incorporate them into a game. However, these early tool sets—such as *UnrealEd* for the *Unreal* series, or the former *Worldcraft* editor for the *Quake* and *Half-Life* series—provided only basic functionality, being restricted primarily to the placement of created objects.

More recently, tool sets have become more powerful (even being released to the public) and often can be used to create the majority of quest and environmental content in a game. The *Elder Scrolls* series, in particular, has gained recognition for its *Elder Scrolls Construction Sets*, which allow a significant level of customisation: terrain forms can be created, non-player characters (NPCs) can be distributed along with their associated quest and dialogue trees, items can be placed and modified, and so on. Similarly, the Warcraft III level editor provides the capability to create full scenarios, including environmental, NPC, and quest editing. While these editors, and others like them, provide a much more powerful and complete game-creation environment, they nevertheless suffer from the same scalability problem: as games become larger and more detailed, increasing amounts of resources are still required in order to produce an acceptably large and detailed world.

2.2 Games and Procedural Content

Hardware limits encouraged many of the earliest games to incorporate some measure of automated content creation. First, because the content to be displayed was relatively simple (basic 2D maps, or, at their most complex, flat ground textures populated by sprites—see the discussion of *Daggerfall* in the previous chapter), concerns of absolute realism were minimal: rectangles joined by meandering lines sufficed as dungeons, for example; or the use of tile-based maps somewhat forgave abrupt changes between neighbouring areas. Secondly, however, size constraints on distribution media and on hard disk space actually discouraged the use of massive, hard-coded environments. Insead, new environments could be generated at run-time—the code to create the procedural content taking orders of magnitude less space than then content itself

would have done. Thus, many early games were quick to take advantage of procedural content, either for data amplification or novel gameplay.

Daggerfall, mentioned above, exemplifies the use of PCG for *data amplification*, a process whereby a finite amount of art assets can be replicated an arbitrarily large number of times. In the case of *Daggerfall*, wilderness areas are "filled in" with random distributions of vegetation and enemies drawn from libraries particular to a given region. Many games use data amplification similarly, in order to flesh out non story-specific areas with content that nevertheless mitigates player tedium.

The use of PCG for *novel gameplay* has its roots in very early games, such as *Elite* or *Nethack*, and is carried on most predominantly in the current generation by the *Civilization* series. In each of these games, and others like them, the emphasis is not on the size of the environment to be explored, but rather the novelty of an environment each time a new game is begun. Using procedural techniques, entirely random and unique planets, dungeons, or landscapes can be generated at each iteration, extending the lifetime of the game by constantly providing new challenges.

As outlined in Chapter 1, procedural content editors have existed in video games since at least 1985. The *Adventure Construction Set* (ACS) released by Electronic Arts allowed a random adventure to be created by the computer, and then modified by an end user [24,30]. While this type of hybrid construction kit has become rare if not nonexistant in the current video game industry, it nevertheless serves as inspiration for Arda system to be developed here.

Approaches to procedural game creation have also been advanced in academia, although generally in the context of more traditional games, such as board games (Chess, Checkers, Go) and card games (Poker, Solitaire). Romein et al. and Pell both introduce game-definition languages—*Multigame* and *Metagame*, respectively—for specifying abstract rules, values, and goals for a game [18,23]. Both systems then use compilers to perform transformations on the defined games: Multigame parallelises as much as possible and proceeds to analyse the set of possible moves in order to build a game-tree search space, while Metagame rapidly creates a playable version of the game for testing. Orwant proceeds in a similar fashion with his Extensible Graphical Game Generator (EGGG), which uses a high-level language to define a

game, and then procedurally creates a playable, graphical version for demonstration purposes [16].

2.3 Procedural Content Algorithms

The variety of available content generation algorithms reflects the numerous disciplines that have been involved in developing them: not only is there a considerable volume of these techniques, but they can be used to create content of a number of different types.

One can consider terrain forms as being the most fundamental level of content common to nearly all games, notwithstanding those taking place entirely in artificial environments—that is, they are generally created first, and changes to the terrain affect any and all objects placed upon it. Accordingly, a large body of work has already been developed, in both video games and visualisation, to automate the process of terrain generation. Mandelbrot, in the mid-1970s, was one of the first to implement such an algorithm, having noticed the fractal nature of many natural landscapes—for example, that the large-scale structure of a mountain range is roughly analogous to much smaller-scale local perturbations [11]. Similarly, Perlin later developed his own Perlin Noise technique, in which similar noise patterns with strictly varying frequencies are added together to provide a global noise function [19]. The latter technique has since become so popular that it is often included by default in most graphics software and video game API's.

Musgrave developed an even greater realism in procedural terrain by introducing two types of erosion into the terrain generation process: hydraulic (erosion by water drainage), and thermal (erosion of rocks naturally breaking and falling down steep slopes) [13]. These forces reduced the the appearance of jagged peaks in the generated terrain, resulting in a more natural look. Much more recently, Olsen defined an *erosion metric* for evaluating the fitness of generated terrains, arguing that these must balance visual/aesthetic interest with actual gameplay (i.e., the ability of structures and characters to be placed on or move around the terrain) [15]. Olsen also introduced

a hybrid technique for terrain generation, combining a base noise function (such as fractal or Perlin noise) with Voronoi cells and pixel-wise perturbations to produce a quick algorithm capable of generating visually interesting and useable natural terrains.

Once a terrain model is built, it is generally desirable to populate it with vegetation, in order to avoid having an overly barren world. In fact, some of the earliest work in content generation was directed towards the generation of natural tree-like structures—this was Lindenmayer's work in the 1960s on the subject of L-systems [9]. Similar to context-free grammars (CFG's), L-systems are a mechanism for string manipulation given a specific alphabet and production rules (which may be deterministic or stochastic). Applying the production rules over a number of iterations, and allowing the underlying symbols to have physical representations (e.g., branching points, branching angles, lengths, etc.), Lindenmeyer was able to replicate structured, tree-like organic entities with a remarkable accuracy.

Lindenmeyer further developed his work alongside Prusinkiewicz, modeling herbaceous plants with great success [22]. They extended basic L-systems by making them more context-aware; for instance, they allow for "genetic" information to be passed down particular branches, as well as between neighbouring "cells". These modifications bring an element of time-dependence to L-systems—whereas the first implementations were directed simply towards an end result, this enhanced version allows structures to be viewed in developmental stages, increasing the realism of the output.

The actual distribution of plant models (themselves procedural or manually-created) can be automated as well. Both Deussen and Wells have achieved some success in simulating complex natural landscapes using L-systems and land cover classification (LCC) bitmaps, respectively [3, 29]. Both techniques incorporate rules for the various species (i.e., models) of plants: min/max elevation; min/max slope; relative elevation; and so on. However, Deussen's L-system technique relies on a number of simulation iterations—replicating competition and migration in ecosystems—to build up the vegetation distribution over time; in contrast, Wells' technique uses the LCC bitmaps as guidelines for random, one-pass distribution. Thus, the L-system approach is more variable at the cost of programming complexity, while the LCC system provides a simple algorithm that is dependent on user input (either from an

accurate geological survey of real locations, or "painted" by an artist).

Parish and Müller extend the use of L-systems even further, noting the similarity between previous L-system content—such as trees and branches, or veins in a leaf—and city-wide systems of roads. Recognising the suitability of L-systems for modelling urban road networks, the authors present impressive results [12, 17]. Their implementation is extended, however: first, the L-system is "self-aware" in that it can form closed loops with itself (i.e., new roads can create cycles by connecting with pre-existing ones); and second, the L-system is highly parametrised—for instance, it can be made to follow population density or terrain elevation, or follow more grid-like patterns as opposed to radial ones.

Meanwhile, research from sociology and civil engineering has presented other unique approaches to urban simulation. Lechner and Watson employ an agent-based technique: a collective of independent "developer agents" is built, in which each individual is assigned a specific role—residential or industrial development, road extension, and so on. Each agent type is endowed with specific goals and biases; when the agents are subsequently "released" into the environment, cities and road plans emerge from their competition for virtual real estate and resources. While this algorithm quite accurately models the dynamic growth of cities under competing forces, the authors admit that the results are coarser and on a much smaller scale than desired [8]. Furthermore, as in any agent-based approach, results are highly unpredictable, though Watson has suggested means for users to significantly influence the final product by placing strong attractors (or "honey") in various areas of the game world [28].

Once suitable procedural road plans have been generated, it is of course necessary to populate these with buildings—much as the generation of random terrain led into the generation of overlaying vegetation. Parish and Müller populate their L-system-generated road plans in a two-step process: first, building geometry (on allotments created during road plan generation) is evolved using another L-system; and second, the geometry is textured using a texture synthesis technique incorporating standard building elements [17]. As with many of the algorithms presented in this chapter, these approaches are capable of impressive results at the expense of programming

complexity. Simpler alternatives are discussed in further chapters.

Other researchers have developed different approaches to generating building geometries, and texturing the blank façades. Greuter begins with a basic (and arbitrary) geometric figure—a square, a hexagon, etc.—and then extrudes the figure downwards in three dimensions; at several intervals, new geometric figures are unioned with the base shape, creating larger and more varied floor plans as the building is extruded from its top down to ground level [6]. Wonka provides an implementation for building façade creation (i.e., the second step of building generation) that is quite similar to L-systems and the CFG's on which these are based. This algorithm defines a *shape grammar* in which basic geometric shapes (i.e., axioms) are replaced by more detailed components according to pre-defined production rules [34].

Proceeding inside the generated buildings, a number of algorithms have been suggested for the generation of floor plans—that is, subdividing a building space into rooms, hallways, and so on. Noel presents an algorithm for iteratively subdividing space into separate polygons, which are then resized according to pre-defined constraints [14]. Hahn provides a similar subdivision algorithm, but extends it to be capable of building-wide constraints (e.g., elevators, stairwells, etc.), as well as suitable hallway distribution (ensuring all rooms are adequately connected to a hallway) [7]. Hahn's algorithm also allows for lazy, "on-demand" generation, reducing the overhead for content creation in general.

While the discussion in the following chapters focuses primarily on the generation of terrain, cities, and building façades, the sheer number and variety of algorithms listed above demonstrates the potential for generating vast and detailed environments, from the underlying landforms to the placement of rooms in buildings.

Chapter 3
Arda System

3.1 Motivation

The previous chapter presented a wide variety of techniques for content generation at various levels of detail—from large-scale terrain masses, to the placement of individual rooms and hallways in buildings. However, as outlined in Chapter 1, these algorithms are generally used in isolation, or strictly for visualisation purposes. The Arda system presented here is an agglomeration of these disparate algorithms for procedural video game content creation. It provides a means of creating different content types, or *content elements* (terrain, cities, buildings, etc.), via any number of interchangeable *generation modules*, wrappers providing a standardised interface to a particular content generation algorithm. Such a system allows researchers to test out and integrate new content-generating techniques within a well-defined framework, and furthermore leads to a tool that can be used by game designers for the rapid, customised creation of large amounts of environmental content that might otherwise be prohibitively expensive to produce manually.

As outlined in Chapter 2, there is a plurality of content types to be generated by the various algorithms that have been discussed; accommodating each and incorporating them into a single tool is an ambitious goal. For the sake of simplicity, the current discussion about—and implementation of—the Arda system is limited to the levels of terrain, cities, and buildings. However, the implementation of these levels

should adequately serve to demonstrate that the larger systems are possible; including further content and module types is simply a matter of repeating the same design patterns that exist within the system already, and is discussed in Chapter 6.

3.2 Overview

The remainder of this chapter presents the Arda content generation system in detail.

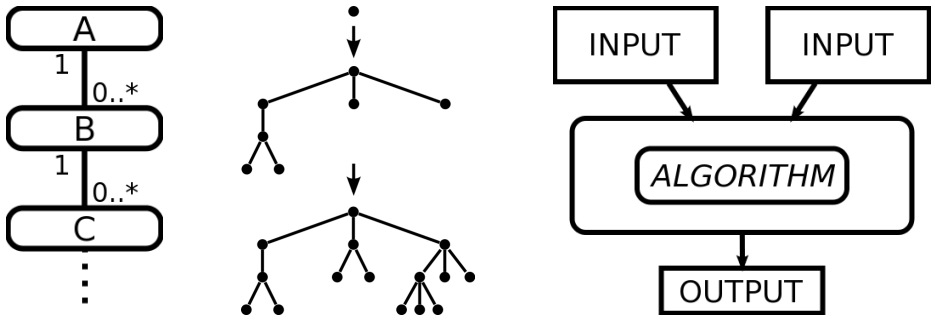
Section 3.3 describes the design of the Arda Framework; that is, the means of organising static content data as well as allowing generation of that content via arbitrarily many different algorithms. Subsection 3.3.1 motivates the need for a data hierarchy, using well-defined, high-level representations of each content type. Subsection 3.3.2 describes the process of procedurally creating environmental content element by element, resulting in a tree-like hierarchy. Subsection 3.3.3 outlines the means by which new algorithms can be integrated into the Framework for use in generating content. Finally, Subsection 3.3.4 describes a remaining pair of components required to complete the definition of the Framework.

Section 3.4 describes the Arda Toolkit, a graphical tool that makes use of the Framework internally, and that allows users to quickly select and customise content generation modules for rapid prototyping, testing, and creation of environmental content. Subsection 3.4.1 describes the various components of the Toolkit's graphical user interface (GUI). Subsection 3.4.2 lists a number of default modules that generate very basic content elements; these have been implemented in order to provide a minimal functionality to the Toolkit. Finally, Subsection 3.4.3 outlines the means by which generation modules in the underlying Framework can be parametrised by a user of the Toolkit front-end through the use of parametrisation modules.

Finally, Section 3.5 provides a summary of the Arda Framework and Toolkit, and concludes the chapter.

3.3 Arda Framework

The *Arda Framework* is the internal software system used to manage both content-generating algorithms, and the actual content data that these create. It is an agglomeration of three logically distinct components: a static data storage structure; a pipeline for creating content using interchangeable algorithms; and an application programming interface (API) affording designers or researchers the ability to develop new algorithms and integrate them into the Arda system with a minimum of effort. These components are shown in Figure 3.1.



(a) (b) (c)

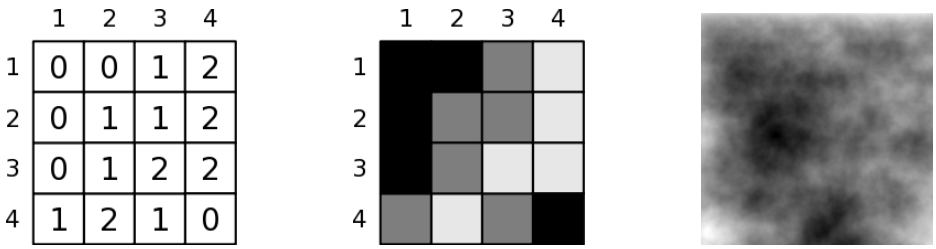
Figure 3.1 – **Overview of Arda Framework components**. (a) Several formalised content data types, arranged in a hierarchy. (b) The process of generating content data. (c) A standard API for content-generating algorithms, to facilitate integration and interchangeability in the Framework.

3.3.1 Storing Content Data

The Arda system is designed to use any number of arbitrarily interchangeable algorithms; it is therefore vital that each one has the same representation of the content being generated. This motivated the first component of the Arda Framework: a static data storage structure, the means by which the Framework represents content internally.

Chapter 1 presented modularity as one of the primary goals of the Arda system, while Chapter 2 outlined the sheer variety of algorithms that could potentially be added to that system: Perlin noise or real geographical data might be used to create terrain; L-systems or cellular automata might be used to generate cities; and so on. Consequently, every discrete piece of content—a terrain, a building, or *content elements* in general—may potentially be generated by a multitude of very varied techniques. To guarantee that any two algorithms generating a particular content element provide data in identical formats, the internal data component of the Arda Framework must define and enforce specific content *types*, or structures. However, while these content types must be strictly defined to avoid conflicts between algorithms, the definitions themselves must nevertheless be abstract enough to accommodate the plurality of content-generating algorithms and the variety of different representations they may use. As the current implementation of the Arda system is responsible for three distinct content types—Terrain, Cities, and Buildings—we define appropriate structures below. Any algorithm generating one of these content types is subsequenty required to create content matching these definitions, guaranteeing consistency regardless of the technique used.

Terrain



(a) Height map in memory

(b) Corresponding B&W image

(c) Typical large-scale height map

Figure 3.2 – **Height maps** as represented in the Framework ((a) and (b)), and in a typical application (c).

Terrain data is stored in the Arda Framework as a height map, a two-dimensional array of numbers whose values correspond to the elevation at a specific point. This can be stored as an actual array internally, or exported to a greyscale image for simple visual inspection or use in external applications—many 3D graphics engines accept such height maps in generating terrain (see Figure 3.2(c)). Note that other applications might also accept or require terrain to be provided in a standard 3D model format (.3ds, .obj, etc), or as an explicitly (i.e., programmatically) defined triangle mesh. However, given a height map, it is trivial to convert to the other formats, and thus the array of height values remains the chosen representation.

Given the simplicity of the terrain concept, and the limited range of probable queries to be performed, designing an abstract interface for terrain items is relatively straightforward. As suggested above, the internal representation chosen for terrain objects in the Arda Framework is simply a two-dimensional array of floating-point values. Queries can be performed on the terrain objects in the same way as for standard array-access: by providing vertical and horizontal co-ordinates, and receiving the corresponding value from the array.

Figure 3.3 – **Interface for a Terrain object**. Terrain is a relatively simple object with which to interface.

Cities

Defining an abstract city data type is a great deal more complicated than the case of terrain. Cities can contain a large amount and variety of semantic information depending on the particular game being played; *Civilization* might record a city's location, size, and population, along with gameplay-dependent factors such as culture, pollution, satisfaction, production, etc.; *Oblivion* may record a city's location, visual style, scale or level of development; the *Grand Theft Auto* series might use information

on which areas are urban or suburban; and so on. However, the Arda Framework is intended primarily for the purposes of *environmental* content generation, allowing us to ignore game-specific considerations and simply consider the physical components of our abstract cities:

- *Position*—The global position of the "centre" of the city relative to some global reference point.
- *Radius/Size*—The approximate extent of the city around the city centre.
- *Roads*—A collection of line segments representing the roads present in the city.
- *Parcels*—A collection of polygons which are defined as the spaces between the constituent roads; these are equivalent to undeveloped land.

Note that while the terms "centre" and "radius" are used, this does not imply that every city must be circular in shape. These are merely approximations defining the position of the city in the virtual world and the extent of its mass; the various city-generation algorithms used might generate any number of city shapes (circular, square, oblong, arbitrary), but each should, in theory, remain concentrated within roughly the same physical area.

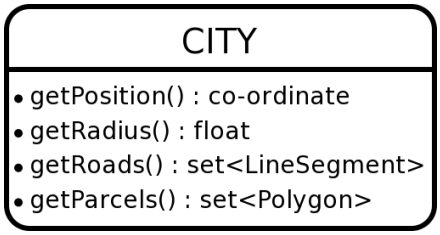


Figure 3.4 – **Interface for a City object**. The queries are more general than those for Terrain; they apply to the whole City, rather than a specific point.

Buildings

Buildings, like cities, are also not easily abstracted, and often require drastically different implementations based on the engine in which they are used. Some games

(such as *Oblivion*) use pre-defined 3D building models placed by hand at specific locations in the game world in a "cookie-cutter" fashion (perhaps linked by an index to a library of building models), whereas others (such as the latest *Grand Theft Auto* games) feature a much more detailed approach in which each building is modelled individually and stored discretely. Still other games, such as Mammoth (a video game project at McGill University) build structures in a much more component-wise fashion, placing each wall, doorway, floor, window, and so on separately. With such a variety of possible building implementations, it is difficult to specify an internal representation for the Arda Framework that will be sufficiently abstract to accomodate all possibilities. Presently, the following elements make up a building structure:

- *Name*—An optional identifier, to allow for indices to be used as mentioned above.
- *Allotment*—A polygon representing the entire area belonging, or available for development, to the building in question.
- *Model*—A specific 3D model in a known format (.obj, .3ds, etc). Also optional, depending on the method chosen to represent buildings in the targeted game.

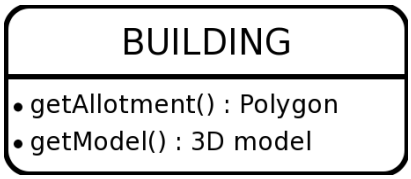


Figure 3.5 – **Interface for a Building object**. These methods are designed primarily for visualisation purposes.

Using the representations listed above guarantees consistency between interchangeable generation algorithms. The Framework then stores these static types as a "content hierarchy," with high-level, large scale content (in this case, Terrain) at the highest levels, and smaller-scale content (i.e., Cities and Buildings) at lower levels. This derives from the observation that most virtual environments contain an implicit set of "one parent, many children" dependencies: for instance, a specific city belongs

to exactly one landscape, but the city itself may contain an arbitrary number of lower-level component buildings, as illustrated in Figure 3.6. The content hierarchy is therefore explicitly maintained in the Framework, where any data element is responsible for storing (and making accessible) its child elements. For example, in this context a single Terrain exists, storing a heightmap as data, along with a set of all child elements (Cities); each City stores its own position, radius, and so on, along with a set of all child elements (Buildings); and each Building stores its own allotment and 3D model, along with a set of all child elements—which, since the current implementation of the Framework consists only of the three listed elements, must necessarily be empty. Accessing a specific allotment, for instance, becomes simply a matter of navigating the content tree from the Terrain, to the appropriate City, to the appropriate Building.

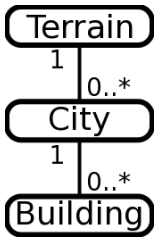


Figure 3.6 – **A view of the Arda Framework's data hierarchy**, demonstrating the "one parent, many children" structure. A single terrain may contain an aribtrary number of cities, which may in turn contain an arbitrary number of buildings.

Finally, it is important to note that while these data abstractions should allow for the majority of game configurations, it is nevertheless possible that they are not sufficiently representative. In the Future Work section of Chapter 6, we discuss the possibility of making content elements more flexible and open to developer customisation.

3.3.2 Content Generation Pipeline

The second component of the Arda Framework is the content pipeline, the process by which environmental content is actually created. The process is tailored to the

hierarchical internal representation of content data outlined above, and allows each new content element to be generated by any suitable algorithm.

While the static data hierarchy reflects the way content data is stored internally, this hierarchical construction is equally useful when attempting to visualise the actual process of creating content within the Framework. Content is created in a top-down approach: for instance, first the Terrain is generated, followed by a number of child Cities, each of which in turn generates multiple child Buildings. The creation of this content tree is strictly ordered: *no content element (besides the root) may begin generation until its parent in the hierarchy has completed its own generation.* This is particularly important given that many of the content-creation algorithms already discussed can make use of information at higher levels of the Framework's data hierarchy: for instance, Parish and Müller's road generation algorithm (along with many others) can be influenced by the underlying terrain shape [17]; likewise, one can imagine building-generation algorithms that reflect the higher-level city in which they are located, perhaps adapting to local road density or other metrics. With a top-down data hierarchy, logical data dependencies are maintained not only in static storage (as previously), but also during the process of content creation. Enforcing this rule allows content generation to occur at any level with the guarantee that all higher-level data is completely generated and available for querying, as in Figure 3.7.

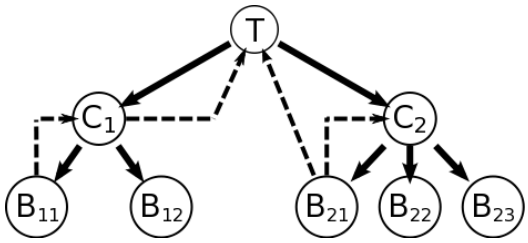


Figure 3.7 – **Querying higher-level content**. Dotted lines show *allowable* queries by content generation algorithms of higher-level, fully-generated data. Here, building B_{11} and city C_1 both query their direct parents in the hierarchy; building B_{21} queries its parent city, along with the highest-level content available, the terrain. These types of queries are made possible by the tree-like hierarchy of data generation. Note that, for instance, building B_{21} could not query city C_1, as they are considered mutually independent in the tree.

In addition to this latter guarantee of higher-level content, we impose the constraint that two content elements on the *same* level (for instance, two cities within one terrain, or two buildings within a city, and so on) cannot share information during the generation process. This entails an important observation: there can be no synchronisation issues between any algorithms operating in the hierarchy. This in turn allows content generation to be completely parallelised within the Arda system; every node in the content hierarchy can be generated within a dedicated thread, with the guarantee that all ancestral data will be complete and available, and that no other module will interfere. An example is shown in Figure 3.8, in which not all nodes are generated at the same speed; however, the top-down approach to content generation ensures that no node is generated out of order. Planning for such parallel processing enables a huge performance boost on modern multi-core and distributed systems, which will be increasingly important as generated worlds grow larger and more complex (i.e., containing more nodes in their corresponding content hierarchy). In Chapter 5, we examine the benefits of the Framework's parallelisation.

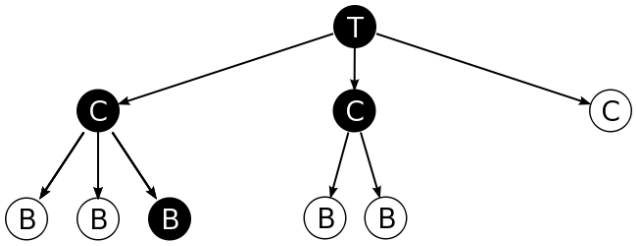
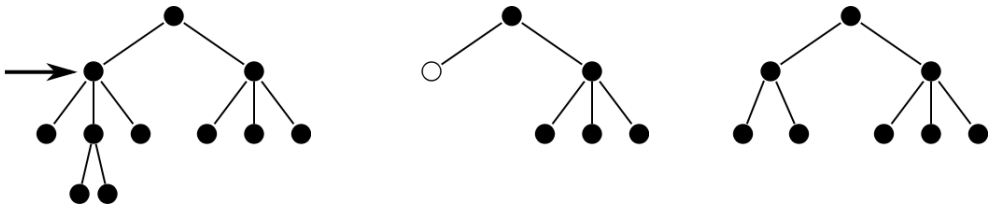


Figure 3.8 – **Content generation in parallel** allows for a drastic reduction in running time, particularly on multi-core systems. Here, completed data nodes (in black) and those whose content generation is currently ongoing (in white) are found simultaneously on any level. Due to assumptions in the Arda system's hierarchy, this parallelisation occurs without risk of synchronisation problems.

Finally, there may be occasions when certain elements within a fully-generated world or hierarchy must be modified or re-generated. Just as the entire content-generation pipeline can be parallelised relatively simply, recreating a given content element is permissible without having to modify or destroy any other content on the

same level—none of the other nodes will be affected, due to previous assumptions about horizontal independence within the hierarchy. However, it must be noted that the *vertical* dependencies present in the hierarchy require, in such an event, that all content below the modified element be destroyed—for example, if one is regenerating a city, it is useless to retain the constituent buildings (and their contents), since the city's road network structure will be completely regenerated. In effect, this implies completely pruning the static content hierarchy at the desired node, and recreating the node itself and its descendents anew. This process is illustrated in Figure 3.9.



(a) Cut (b) Regen (c) New

Figure 3.9 – **Regenerating content at a node**. The entire subtree located at the node to be regenerated is pruned in (a) and (b). A completely new subtree results from subsequent regeneration (c).

Difficulties

The strict hierarchical, tree-like requirements of the Arda Framework are not without drawbacks, however. The situation outlined above, in which the regeneration of one node results in the discarding of an entire branch of the content tree, is potentially sub-optimal in that it may involve a great deal more computational power to regenerate the branch than to simply change the individual node itself. While a final, static, exported environment can be manually modified (e.g., an artist can move roads or reshape buildings) without destroying a large amount of content, the Framework as currently implemented does require full branch regeneration in order to maintain data dependencies and allow the queries described in Section 3.3.2. Furthermore, there may

be algorithms for content generation that operate on several elements simultaneously: for instance, an algorithm for Terrain- and City- generation which relies on feedback between the two elements; or two City elements that share a common highway, and thus contain similarly-oriented road patterns. Such possibilities are currently ruled out in the Framework implementation.

Both of the above situations are caused by the strict hierarchical structure used in the Arda Framework. Though this hierarchy enables a vast amount of parallel computation, other structures could allow a wider range of interaction between elements in the Framework. Such possibilities are discussed further in Chapter 6.

3.3.3 Generation Modules & Extending the Framework

The third component of the Arda Framework is the application programming interface (API) allowing any procedural content-generation algorithm to be integrated into the Framework itself.

The two previous sections have outlined the abstract content representation internal to the Arda Framework, as well as the overall process for generating content within that hierarchical structure. However, the primary goal—and major asset—of the Arda system in general is the ability to use any appropriate algorithm to procedurally generate content anywhere within the hierarchy. This requires not only the formalised data structure presented in Section 3.3.1 (which ensures that any two algorithms creating the same type of data have identical representations of that data), but also a means of using algorithms interchangeably without affecting any other components in the content-generation pipeline—for example, swapping Perlin noise for midpoint displacement to perform terrain generation should have no effect on a lower-level city generation event using cellular automata. To achieve this, we modify the process of generating content at multiple levels to allow for "black boxes" at any point that would otherwise require human input. Strictly speaking, these black boxes are interfaces in the object-oriented programming sense—placeholders that delegate function calls to any designated object that follows a particular signature. In this case, the objects in question are called *generation modules*; a given generation

module wraps around a particular content-generation algorithm, and can then be "plugged into" the appropriate interface in the content-generation hierarchy. When called upon, they generate exactly the content required, in the appropriate format, and then cede control back to the main content pipeline. Figure 3.10 demonstrates this process: here, a Terrain interface allows any one of several generation modules to generate the Terrain data; each module itself wraps around a particular algorithm.

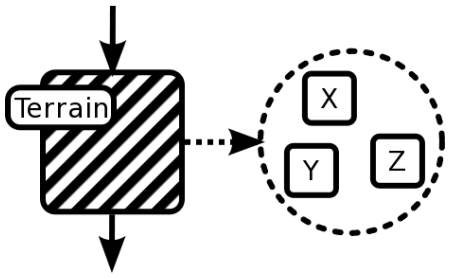


Figure 3.10 – **A module for generating terrain**. The "black box" interface for Terrain generation allows any one of the generation modules X, Y, or Z to be used to generate terrain interchangeably; the rest of the content pipeline remains oblivious to any changes. Each generation module wraps around a particular algorithm, such as Perlin noise, fractal techniques, etc.

As indicated previously, any generation module that is being used in the content pipeline (and, by extension, the algorithm around which it wraps) can make the assumption that all higher-level content has been fully generated; in effect, that content can be queried and used as input for the given algorithm. In addition, some content algorithms require or make available a number of other tuning parameters that influence the final result: Perlin noise uses a specified number of octaves; L-systems may use different rules, and alter production rule probabilities; and so on. These parameters allow developers and researchers to experiment with or otherwise configure the content to be generated. Given the sheer variety of these possible parameter types, the task of specifying parameters is not part of the Arda Framework itself; rather, generation modules must allow for parametrisation themselves (e.g., through the "get" and "set" methods common to object-oriented code), as well as providing

a set of default parameters. When the Framework is embedded in an external application (for instance, as part of a game engine, or as a separate content tool), this application then becomes responsible for interfacing with the generation module and tuning the parameters as required, as seen in Section 3.4. The algorithm implemented by the generation module is then executed, with all the appropriate input and parameters guaranteed. Finally, the generation module provides the generated content back to the content-generation pipeline, in a format consistent with the content types described in 3.3.1. Given all of the above, an abstract representation of a generic generation module is shown in Figure 3.11.

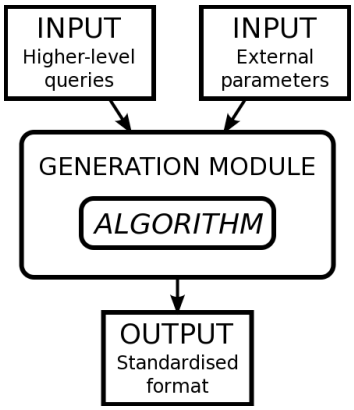


Figure 3.11 – **A generic generation module**. Each generation module is guaranteed access to higher-level content, as well as parameters defined by an external application. The algorithm then executes with this information, and generates the appropriate data.

In Section 3.1, we specified that the current implementation of the Arda system focuses on a particular subset of video game content: terrain, cities, and buildings. As further noted in 3.3.3, the content pipeline requires the use of generation modules at any point where human input would otherwise be needed; thus, the choice of content elements used in this implementation of the Arda Framework dictates the type of generation modules required (and which will be created in Section 3.4). They are listed below, and illustrated in Figure 3.12.

- *Terrain*—Generate the full terrain height map.

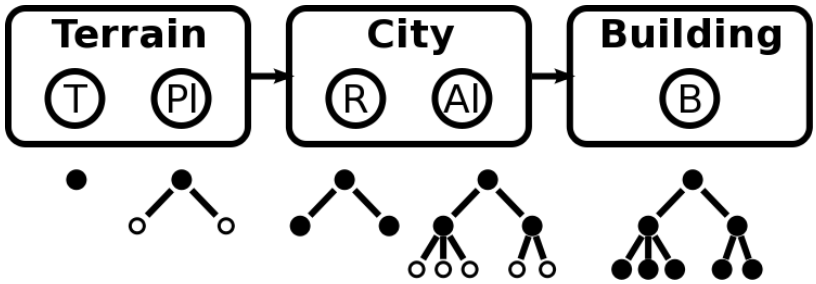


Figure 3.12 – **Current pipeline implementation**. Top: the static elements currently implemented in the Framework and their associated generation interfaces—(T)errain generation, City (Pl)acement, (R)oad generation, (Al)lotment division, (B)uilding generation. Bottom: an example content hierarchy being generated by these modules. Notice how (Pl) is responsible for creating several Cities as children of the Terrain, and (Al) is similarly responsible for creating Buildings for each City.

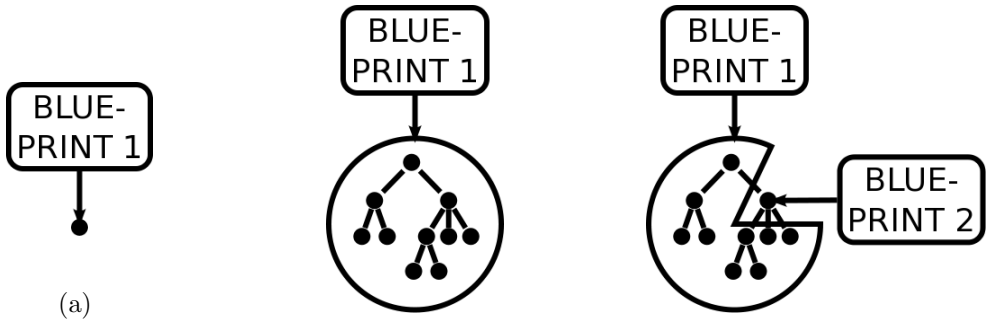
- *City placement*—Determine the number of cities to generate, along with their placement and size. While this might be seen as three separate tasks, the values chosen are inter-related: creating very large cities limits the choices for their number and placement, and so on. Using one module to determine the three values ensures complete control over the balance between these parameters.
- *Road networks*—Create networks of roads throughout each city.
- *Allotment generation*—Divide the large land parcels created in the previous step into individual building allotments.
- *Building façades*—Create and model building exteriors.

3.3.4 Additional Components

A difficulty arises from the use of generation modules in the dynamic, arbitrarily-defined content pipeline. The discussion to this point seems to suggest that the generation modules required for each particular content element are hand-picked for each individual node. Not only would this require human input (something the Arda system is designed to eliminate) for each node, but it also does not make sense, for

instance, in the case where the entire content hierarchy has not yet been generated: no content elements yet exist, and so no generation modules can be selected. Instead, a particular set of generation modules—one for each available interface, i.e., five in the current implementation of the Framework listed above—is selected before the content pipeline is executed. This subset of modules is called a *module blueprint*, and defines which modules are to be used at each level of content generation. Given a module blueprint, a particular content element can determine which generation modules to use during the creation phase; furthermore, every child of that node inherits and makes use of the same module blueprint for their own generation. Both new generation and inheritance using a given module blueprint are shown in Figure 3.13(a) and (b). In addition, once the full content generation process is complete, individual nodes may be selected, and their module blueprints modified; thus, in the case of regeneration from a particular node (as in Figure 3.9), an entirely different set of generation modules—as opposed to those chosen initially—can be used to regenerate the given branch, as shown in Figure 3.13(c). The introduction of module blueprints therefore addresses the problem of specifying generation modules for non-existent content elements, as well as removing the need for human input (i.e., the selecting of modules) during generation itself.

Finally, a *module manager* is added as a means of managing the use of generation modules within the Framework. This component allows new generation modules to be added to the Framework, while avoiding duplicates. Furthermore, it ensures that no generation module is assigned to an incompatible point in the content generation pipeline (e.g., a module for building allotment generation being assigned to a terrain node). Lastly, we note that when a given content element is created (for example, a new Building is defined), it may share the same *type* of generation modules as other similar elements—for instance, two Cities may use L-systems for road generation—but they must not share the same *instance* of a generation module, since it is quite possible that a user might want each city to be parametrised differently. The module manager is therefore responsible for instantiating new generation modules for each newly-generated content element, ensuring that each content element, and its associated generation modules, can be parametrised individually.



(a) (b) (c)

Figure 3.13 – **Distribution of module blueprints**. In (a), a single element is assigned a blueprint containing a list of modules to use during generation. (b) shows the content hierarchy after generation; all elements were generated using the modules indicated in Blueprint 1, and have inherited the blueprint itself. (c) shows a single node having been assigned a new module blueprint; if generation is begun at that node, the sub-tree is destroyed and modules from Blueprint 2 are used in regenerating it.

The combination of the above components completes the design of the Arda Framework. Content types (in the present implementation: "Terrain," "City," and "Building") abstractly represent the data that can be generated, and are stored hierarchically. The content pipeline follows this hierachical pattern, creating a tree of content elements; each branch of the tree is generated node by node, and every branch in parallel. The actual generation of a content element at a particular node is left to generation modules, objects that implement a given content-generation algorithm, and obey strict input-output signatures. Once created, the generation modules are added to the module manager, which itself is called upon during content generation to provide appropriate modules where required. Lastly, users can use module blueprints to define, at any node of the content tree, which modules will be used for subsequent content generation. The design of this Framework allows many different types of content—which, until previously, had been generated strictly in isolation—to be brought together into a single representation, and generated by any number of applicable algorithms, as many as programmers wish to implement.

3.4 Arda Toolkit

The Arda Framework presented above combines a hierarchical data structure with what is essentially an application programming interface (API) allowing game designers and algorithm researchers to "plug in" their own generation modules according to specific object interfaces. The Arda API and libraries are designed in such a way as to be self-sufficient: they form a closed system that can function independently from any external application, and the resulting content can be exported to any format required. This flexibility affords the Arda Framework the possibility of being integrated into any number of applications making use of automatic content creation; for instance, it could be embedded within an actual video game in order to make content generation a distinct part of gameplay, or it can be used as the principal API within a software tool specifically designed to generate video game content. The latter approach was used here, leading to the development of the *Arda Toolkit*, a graphical user interface (GUI) that can be used to simplify the process of creating procedural content for games; a user can intuitively import newly-implemented generation modules into the underlying Framework, select generation modules to be used during content generation, and can quickly evaluate the resulting content and adjust parameters or switch algorithms appropriately.

3.4.1 Design

The Arda Toolkit is comprised of three components: the panel, the view, and the model.

The *panel* presents to the user a tree-like outline of any content that has previously been created. Within this tree, the user can select any number of nodes (i.e., content elements) to be regenerated or re-parametrised. Note that, if a node and one of its descendants are selected for regeneration, only the node itself is regenerated; all lower-level content is deleted before being recreated, thus eliminating the lower-level node entirely (see Figure 3.9). Furthermore, when a content node is selected in the outline, that element's module blueprint is presented to the user, with each generation module

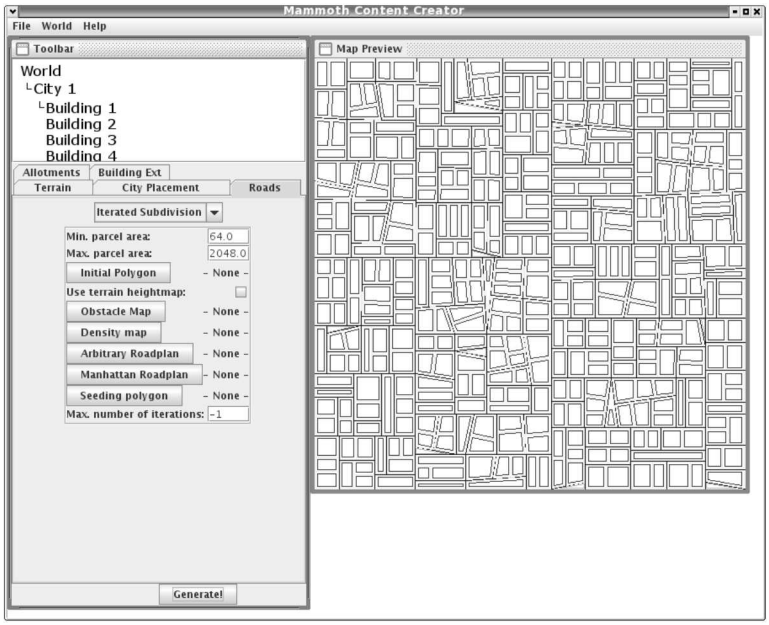


Figure 3.14 – **Current implementation of the Arda Toolkit**. On the left, the "panel" contains the current content hierarchy, as well as a parametrisation tab for each of the currently-selected generation modules. On the right, the "view" displays a simplified version of the content created thus far.

given a separate tab. The user may then choose which module to use at a given level of content creation, or reparameterise one that is already present. The panel is shown in Figure 3.14—note both the tree-like content outline, and the modules available for parametrisation.

The *view* is a small window displaying the state of the current world. In simplified form (a two-dimensional projection of a fully 3D world, for instance), the view provides immediate feedback on the extent of content creation, the capabilities of a particular algorithm, or the effects that changing parameters may have on the resulting content. The view is shown in Figure 3.14 after content creation has occurred and both a City and its component Buildings have been generated.

Finally, the *model* is simply the aforementioned instance of the Arda Framework that underlies all Toolkit operations. As users of the Toolkit add and select generation modules, they are added to and activated within the Framework, respectively. When

parameters of a specific module are changed via the Toolkit, the underlying model is changed as well. Likewise, when a content generation phase is initiated through the Toolkit, all of the appropriate functionality is performed within the Framework itself. In this sense, the Toolkit is effectively a visual representation of the Arda Framework with a much simpler (i.e., graphical) interface to an external user.

3.4.2 Default Generation Modules

The intent of the Arda Toolkit is that a novice user can run the program and begin generating content within minutes, with little or no parametrisation required. However, the Framework defined above merely presents interfaces for generation modules—"black boxes" into which algorithms can be inserted—without providing any underlying implementations. In order for the Arda Toolkit to be able to generate a minimal amount of content, at least one generation module must be implemented for each available interface, to allow the content pipeline to operate fully. In 3.3.3, we defined the interfaces specific to the current implementation of the Arda system, as shown in Table 3.1; we must therefore implement at least one default generation module for each of these interfaces. The modules themselves are defined below—however, note that they perform the bare minimum of content generation, with little or no regard for realism or aesthetics, and act primarily as place-holders for future algorithms. More realistic implementations for some these modules are discussed in the next chapter.

Terrain: Flat Surface

The most basic method of generating terrain is to simply generate a completely flat surface—i.e., a height map or grid of identical values. While not very visually appealing, it allows designers and developers to focus instead on other parts of the content pipeline, such as generating road networks or building façades. Despite this technique's simplicity, however, we allow users to specify a height value for the terrain, should such information be required to better integrate the generated terrain with an existing world, for instance.

Content Element	Interfaces
Terrain	Height Map Generation City Placement
City	Road Networks Allotment Generation
Building	Building Façades

Table 3.1 – **Content elements and their associated interfaces**. The current Arda system defines the three types of content element shown on the left. Each content element requires certain operations to be performed during the course of content generation; these are the interfaces shown on the right which will delegate the operations to the appropriate generation modules.

City Placement: Random Number Generation

In the most basic case, the placement of cities in a virtual world can be left entirely to chance by generating arbitrary world co-ordinates. More sophisticated algorithms, such as those presented by Olsen, might take gameplay considerations into account and only choose locations more suited to cities (such as those with large expanses of relatively flat terrain) [15], but would likely require a measure of pattern recognition beyond the scope of the current discussion. Therefore, unlike terrain generation, the placement of cities is entirely random, and does not require parametrisation.

As with the placement of cities, the selection of a specific city size in our placeholding generation module is both arbitrary and entirely ignorant of gameplay considerations. However, in the case where there are multiple cities to be generated in the virtual world, it may be desirable that they occupy roughly similar amounts of space. For this reason, city size selection is parametrised by a Gaussian distribution whose mean (μ) and standard deviation (σ) can be specified by the developer. This allows a multitude of cities to be generated with an approximate desired size, but also allows for as much variability as required.

Road Networks: Grid Pattern

Given the use of purely arbitrary data in the default generation modules above, it might be tempting to simply generate completely random line segments to be used as roads in this module. However, as will be shown in Chapter 4, this creates unappealing, chaotic road networks, and so another technique is required. One method is to create a simple grid-like pattern of roads: a number of roads that span the city both horizontally and vertically, and which cross each other perpendicularly. The user can specify the number of roads in either direction: at least one horizontal and one vertical road, up to any arbitrary amount within machine limits.

Road Networks: Voronoi Cells

As outlined in Chapter 2, Glass has had great success using randomly-generated Voronoi cells to simulate the haphazard but partially-structured appearance of informal settlements [5]. While Glass' work relates mostly to contemporary "shanty town" settlements in South Africa, it can be argued that the results bear some qualitative similarities to the fantasy-style cities seen in many modern role-playing games (RPGs), such as *Dungeons & Dragons* or *Oblivion*. Such cities are often loosely based on actual medieval cities or towns, which themselves were traditionally also organised informally. This module therefore makes use of the JTS computational geometry package to generate random Voronoi cells to serve as a set of roads and land parcels [27]. For this module, the user can input the number of randomised seeding points used in generating the Voronoi cells.

Allotment Generation: Single Buildings

Road network generation provides a city object with a collection of road segments—simple lines—and a collection of land parcels—polygons that make up the space between roads. Ideally, the parcels should be subdivided so that there are many buildings on one city block, as opposed to one massive building; the former being more common in most urban regions. Again sacrificing realism for the sake of simplicity, this module will simply generate one allotment per parcel, using a geometry package

(again, such as JTS) to create an interior offset polygon, to offset the allotment from the centre of the road.

Building Façades: Simple Blocks

Once again, simplicity is the primary concern of the building façade generation module. This module merely extrudes the building's allotment polygon (its "footprint") in three dimensions, giving the building a height; a random texture is then chosen from a set of building textures, and applied to the newly-created model.

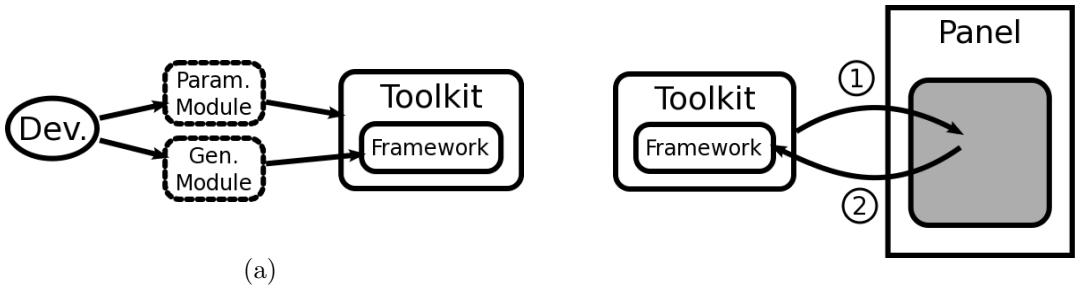
Having defined all of these default generation modules, the Arda Toolkit can be immediately employed by a new user to create content (however uninteresting) without an initial programming task.

3.4.3 Module Parameters

Once a new generation module has been implemented (see Subsection 3.3.3), it can be added programmatically to the set of generation modules in the Framework, and subsequently used at any appropriate content generation step. However, most modules (including several listed above) allow parametrisation of a number of different values for customisation purposes, a process that is not provided for by the Framework itself. Instead, each generation module is expected to expose its parameters for reading and writing by an external application—in this case, the Toolkit. However, the "panel" component of the Toolkit cannot be constructed in such a way as to anticipate all possible parametrisations available to generation modules—these may require any type of input, including text or integer values, sliders, lists, and so on, in any arbitrary number and combination. As such, when a content element is selected in the panel's tree-like view of the data hierarchy, and that element's module blueprint is evaluated to determine which generation modules to display, the panel cannot presuppose which type of parameters to present to the user.

To enable the parametrisation of generation modules within the panel, we require that when a generation module is loaded into the Framework via the Toolkit, a corresponding panel interface be loaded for the Toolkit itself. For instance, if a

generation module for Perlin noise is created, a corresponding graphical interface is provided that allows a user to specify a number of octaves to use during generation; both components are bundled together and provided to the Toolkit, which passes the generation module to the Framework, and stores the required graphical interface internally. When a node is subsequently loaded that has a module blueprint containing the Perlin generation module, the Toolkit is able to identify which graphical interface to use, and prompts the user appropriately for input (see Figure 3.15(b)). Essentially, this means that any designer or researcher who aims to use the Arda Toolkit to make use of new content-generation algorithms must create two components: a generation module—implementing the algorithm itself—for use within the Arda Framework during execution of the content-generation pipeline; and an interface module for the Arda Toolkit, providing a graphical interface to the parameters within the generation module, as in Figure 3.15(a)).



(a)

(b)

Figure 3.15 – **Using parametrisation modules in the Toolkit**. In (a), a developer ("Dev.") creates two modules: a parametrisation module for the Toolkit, and a generation module—wrapping around a content algorithm—for the Framework underlying the Toolkit. In (b), the parametrisation module informs the Toolkit (1) how to present parameters in the panel (see Fig. 3.14); the parameters entered by the user through the panel are then redirected (2) to the appropriate generation module in the Framework.

3.5 Summary

In this chapter, we have introduced the Arda system for procedural content generation, composed of the Arda Framework and the Arda Toolkit.

The Arda Framework is a data structure composed of a number of objects abstracting common content types (terrain, cities, buildings), as well as software interfaces for modules designed to generate that content. Its hierarchical structure enforces a strict ordering during the generation process—terrain is generated before cities, cities before buildings, and so on. This allows generation modules to access previously-generated data at higher levels (as many algorithms do), as well as enabling a vast amount of parallelisation.

The Arda Toolkit is a visual tool or GUI allowing developers to easily make use of the Framework in order to experiment with new content generation algorithms, as well as simply to generate their own virtual worlds. The Toolkit allows users to choose which generation modules will be used at each level in the content generation pipeline, as well as allowing the complete regeneration of the content subtree rooted at any specific node in the Framework. The Toolkit provides visual feedback as well, in the form of a small preview of the content generated at any given point. Once satisfied with the end result, the designer can export the data stored in the Framework to the format of their choice.

A number of default generation modules have been implemented for the Toolkit, to allow it to be used immediately by any designer and avoid a long programming task for new users; however, these default modules are aesthetically quite basic. In the next chapter, we introduce a new algorithm for road network generation, capable of much more sophisticated results than those introduced thus far.

Chapter 4
Iterated Subdivision

4.1 Motivation

In Chapter 3, we looked at the overall structure of the Arda Framework and implemented a handful of very simple generation modules to serve as "place holders" in the content generation pipeline. These basic modules generally use very simplistic algorithms and, while quickly implemented and executed, only generate qualitatively "uninteresting" content that would be unfit for use in a commercial game. Our goal, however, is to show that the Arda content generation system is capable of producing large-scale, complex environmental data in a fraction of the time it would take for human artists to do the same; this requires more powerful algorithms at a number of points in the content-creation pipeline. The current chapter introduces such an algorithm: Iterated Subdivision, a technique that can be used both for city road network generation, as well as allotment subdivision.

4.2 Iterated Subdivision

Several road-generation algorithms were discussed in Chapter 2; in particular, the use of L-systems or agent-based approaches has become relatively common. However, while these have proven successful in a number of ways, they come at the cost of

programming complexity. Large L-systems (e.g., being used to generate large cities) often become slow and inefficient; furthermore, the extension to parametric L-systems (adapting to internal context and external stimuli) adds another layer of programming and conceptual complexity. Agent-based approaches, on the other hand, require a great deal of programming overhead to create a multitude of agent types, while the interactions of the latter are often so complex as to entail a prohibitive amount of parametrisation and fine-tuning. Our goal was therefore to develop an algorithm that—qualitatively—is approximately as expressive and powerful as the two latter approaches, while retaining a conceptual simplicity to make it more accessible to the majority of programmers.

Our approach draws inspiration from Tarbell's *Substrate* visualisation, which was noted to produce "intricate city-like structures" [25]. Tarbell's algorithm places "seed" vectors in a two-dimensional canvas and proceeds to extend these vectors as far as possible; new vectors are added roughly perpendicular to these seeds and any other vectors that have subsequently been generated. Similarly, the algorithm presented here begins with a two-dimensional polygon and proceeds to repeatedly bisect it with deliberately chosen line segments. Since it iterates a (controllably) finite number of times, and involves the repeated subdivision of polygons, the algorithm has been dubbed *Iterated Subdivision*, or `ItSub`.

The initial form of the algorithm requires as input merely a predefined, simple polygon P, and minimum and maximum areas—A_{min} and A_{max}, respectively—for the resulting subdivisions (which in this case represent tracts of land between roads, or "parcels"); further initial parametrisation is possible, and discussed below. The algorithm then bisects P randomly, resulting in two new simple polygons. These are then either accepted or rejected based on their size: large polygons are subdivided further, while those below a certain size threshold are set aside as completely subdivided. The algorithm continues until there are no more polygons to subdivide.

The basic `ItSub` process as described above is formalised in Algorithm 1. The key data structure in the algorithm is $S_{oversized}$, a set containing every (unique) polygon that is larger than A_{max} (i.e., every "oversized" polygon), which must therefore be

Algorithm 1: Basic Iterated Subdivision

Input: Polygon P, A_{min}, A_{max}

Output: S_{parcel}, S_{roads}

```
1  S_oversized ← P
2  S_parcel ← S_roads ← ∅
3  repeat
4      P_working ← S_oversized.poll()
5      repeat create random bisector
6          L_bisect ← random bisector, random angle
7          {p_1, p_2} ← P_working subdivided by L_bisect
8      until |p_1| ≥ A_min and |p_2| ≥ A_min
9      S_roads ← S_roads ∪ L_bisect
10     foreach polygon p_i in {p_1, p_2} do
11         if p_i.area > A_max then
12             S_oversized ← S_oversized ∪ p_i
13         else
14             S_parcel ← S_parcel ∪ p_i
15         end
16     end
17 until |S_oversized| == 0
```

subdivided further. $S_{oversized}$ is initialised with the starting polygon P, and subsequently provides polygons one at a time via $S_{oversized}.poll()$. Every working polygon ($P_{working}$) thus provided is then randomly subdivided. This is done by randomly selecting an edge of the polygon, then randomly picking a point along this edge and drawing a bisecting line, L_{bisect}, through the polygon at this point—note of course that we assume that L_{bisect} is not parallel to the chosen edge. The bisection results in two new simple polygons, p_1 and p_2, which are evaluated individually. If either of these polygons is too small (that is, smaller than A_{min}), then L_{bisect} is discarded, and a new bisector is again chosen randomly. Once a valid bisecting line is returned, L_{bisect}

is added to S_{roads}, the set of all valid bisecting lines, which is interpreted in this case as the set of all roads belonging to the resulting city. Finally, p_1 and p_2 are measured in relation to A_{max}. If a polygon is larger than A_{max}, it is still oversized, and therefore returned to $S_{oversized}$ to be subdivided further; if not, it is added to S_{parcel}, which represents the final set of land parcels resulting from subdivision. Note that while S_{parcel} could be calculated from S_{roads} (and vice versa), it is stored separately here to avoid unnecessary post-processing. As outlined above, the algorithm continues to process polygons in this fashion until the entire space is sufficiently subdivided—that is, when $|S_{oversized}| = 0$.

4.3 Refinements

The `ItSub` algorithm, as presented above, adequately approximates the Substrate algorithm on which it is originally based. It uses a minimum of inputs: a starting polygon, P; A_{max} to ensure that the algorithm eventually terminates; and A_{min} principally for aesthetic purposes, to require that the resulting subdivisions (i.e., "parcels") are above some minimum acceptable size. However, there are a number of improvements that can be made without drastically altering the basic algorithm; these are described below.

4.3.1 Diameter-to-Width Ratio

The selection of bisecting lines is clearly critical to the aesthetic success of the `ItSub` algorithm, and various constraints can be applied to the process to ensure more realistic output. With the naïve implementation of Algorithm 1, degenerate subdivisions can sometimes appear, as shown in Figure 4.1. In this example, a number of subdivisions near the top of the image seem extremely long and narrow; nevertheless, the given polygons are smaller than A_{max} and larger than A_{min}, and are thus considered valid by the basic `ItSub` algorithm. However, such overly-elongated constructions are rare in urban areas, and would be overly constricting on any contained buildings.

To avoid the problem of excessively long and thin subdivisions, an extra constraint

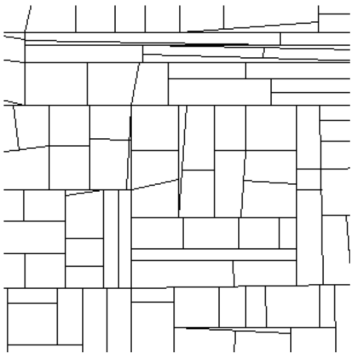
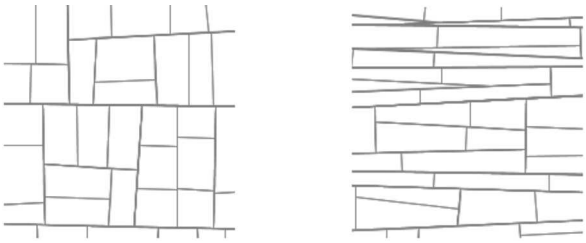


Figure 4.1 – **Overly-elongated subdivisions** are evident at the top of the figure. These would generally be unacceptable in a standard video game.

must be placed on the range of "acceptable" bisecting lines, and the sub-polygons they produce. Previously, it sufficed that a polygon's total area be larger than A_{min}. Now, a new constraint must be placed on the ratio between the polygon's largest and smallest spans—that is, between the polygon's diameter and width, as defined by both Preparata and Toussaint [21, 26]. As this ratio approaches infinity (that is, as the diameter becomes increasingly larger than the width), the polygons become more elongated; as the ratio approaches unity, the polygon is contracted to become more square-like. A bisector is thus defined to be acceptable if the resulting sub-polygons have diameter-to-width ratios below a certain threshold—which we call R_{max}—in addition to areas larger than A_{min}. The effects of varying R_{max} are shown in Figure 4.2; tests have shown that setting $R_{max} = 16$ provides adequate results, and this value has been used in all subsequent figures.

4.3.2 Branching Angles

Returning to Algorithm 1, we note that the angles of the chosen bisecting lines—L_{bisect}—are unconstrained; that is, a bisecting line is allowed to pass through the working polygon in any direction. In practice, however, this leads to completely arbitrary road patterns, as shown in Figure 4.4(b). While a certain degree of arbitrariness is expected in urban road networks, the effect is unrealistic on any large scale; indeed, the majority of (especially modern) urban areas tend to display much more regular,



(a) (b)

Figure 4.2 – **Different diameter-to-width ratios** and their effects on resulting subdivisions. (a) shows a small ratio, and (b) a much larger one.

right-angled road patterns. When a random edge is chosen from a polygon, and a random point chosen along that line, an acceptable solution to the latter problem would therefore produce bisecting lines that are generally perpendicular to the chosen line at the given point—we say that the *branching angle* is 90 degrees—with the possibility of some variation. To achieve this, the current implementation of the `ItSub` algorithm initialises any branching angle to 90°, and then perturbs this angle by an amount determined by a Gaussian distribution, whose mean (μ) and standard deviation (σ) parameters can be defined at run-time. Figure 4.3 illustrates the process.

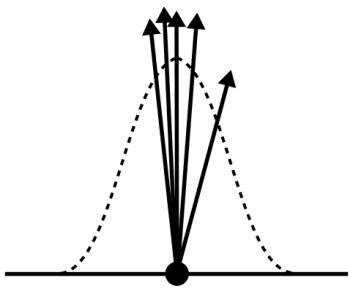
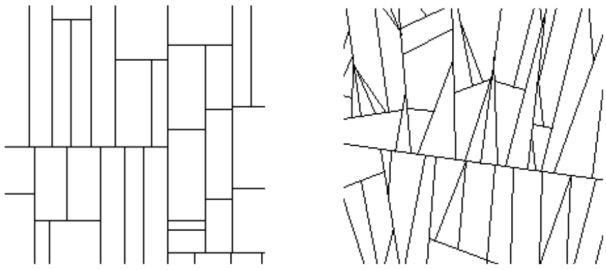


Figure 4.3 – **Branching angles** determined by a Gaussian. The mean perturbation is 0°; larger angles are more rare.

As described elsewhere, many urban regions are composed of more than one distinct "style" of road plan—large-scale patterns emerging from the orientation of a local collection of roads [17]. The manipulation of branching angles as outlined above

allows the `ItSub` algorithm to produce at least two different road plan styles, which have been dubbed Manhattan and Arbitrary. *Manhattan*-styled roads—named after the city in which they are most prevalent—exhibit branching angles that rarely (if ever) deviate from 90°, and are common in modern, commercial, or rigourously-planned neighbourhoods. This style of road plan can be achieved by modifying the perturbation Gaussian above with a μ and σ of 0—thus ensuring that all branching angles are perpendicular, and resulting in the grid-like pattern desired. *Arbitrary* roads, on the other hand, are more typical of older neighbourhoods, in which strict urban planning was less of a concern than simple expedience. In these areas, there is no specific guideline or obvious pattern to branching angles, and the resulting roads tend to be somewhat chaotic; achieving this effect is once again possible by manipulating the perturbation Gaussian's standard deviation. Here, σ is significantly increased (e.g., to 15° or larger), resulting in a wide variety of branching angles, and a more arbitrary pattern of roads. Figure 4.4 shows examples of both of these road pattern types.



(a) Manhattan (b) Arbitrary

Figure 4.4 – **Examples of different road patterns** possible through the parametrisation of branching angles in the `ItSub` algorigthm.

4.3.3 Snap distance

Due to the arbitrariness of the `ItSub` algorithm, it can often be the case that a given bisecting line either begins or ends close to, but not precisely at, a previously-created intersection. When this is the case, it results in a comparatively short road segment—or as we interpret it, a short city block. However, these short "jogs" do not generally occur in the majority of road networks, and so a solution is required to eliminate such constructions from the final road network. We define a new parameter D_{snap}, the minimum distance between intersections in the final road network; when an endpoint of the bisecting line L_{bisect} is determined to be closer to an intersection than D_{snap}, that endpoint is displaced, or "snapped," to the intersection itself. The process is illustrated in Figure 4.5: (a) shows two completed polygons (in grey) meeting at an intersection, and part of a working polygon in white; in (b), the projected endpoint of L_{bisect} is found to be less than D_{snap} away from that intersection; (c) shows the resulting projection when snapped into place; and (d) shows the newly completed polygon (it is assumed that the bottom-left polygon, in white, requires further subdividing). Note of course that both, either, or none of the bisecting line's endpoints may be snapped into place—snapping one endpoint to an intersection does not preclude snapping the other.

4.3.4 Bitmap Parametrisation

Constraining diameter-to-width ratios and snapping together nearby road endpoints (as above) are both modifications to Algorithm 1 that ensure realistic results; they are both applied globally. There are, however, certain aesthetic qualities of the resulting road plans that can be much more localised, and can exhibit a large degree of variability. For such properties, it is desirable to allow user paramitrisation through the use of bitmaps similar to those in Figure 3.2.

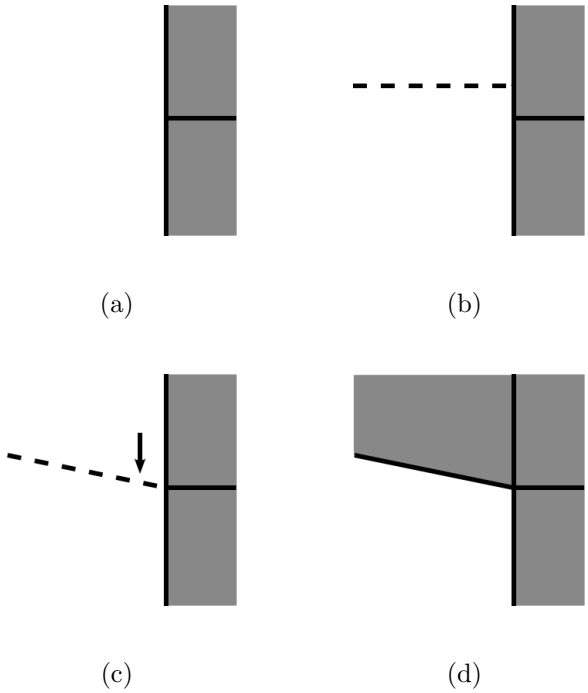
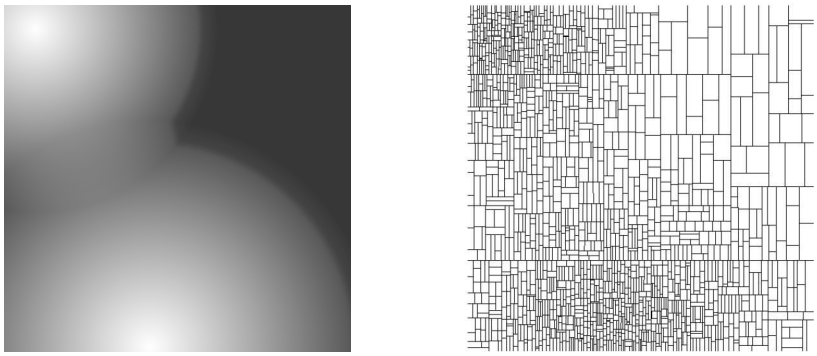


Figure 4.5 – **Snapping road endpoints** eliminates the appearance of small "jogs" in the resulting road networks.

Road Density

Nearly all urban areas exhibit some degree of variation in population and road density across their surface. In the basic `ItSub` implementation, however, a uniform density is implicitly assumed—polygons are subdivided until all are between A_{min} and A_{max}, both invariant values. In order to parametrise the variation of road density, we use greyscale bitmaps to represent the appropriate changes: light areas in the bitmap indicate high road density, darker areas indicate low density.

Algorithm 1 is altered slightly to allow for user-defined density maps. In the base case, where no density map is provided, there is no difference with the original algorithm. Conversely, when a map is provided, extra processing is performed when testing each polygon resulting from a subdivision: the values of A_{min} and



(a) Density map (b) Resulting road plan

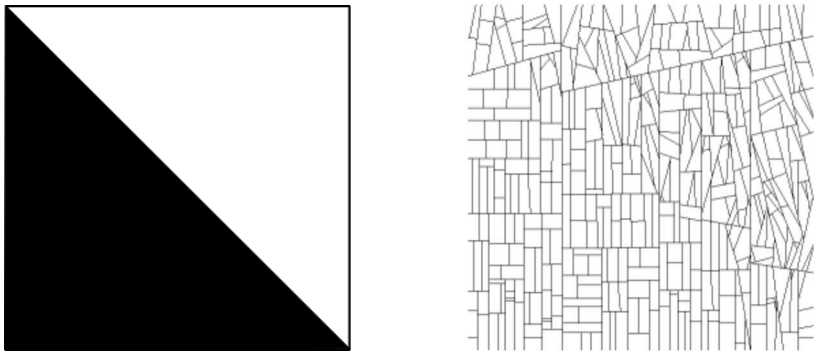
Figure 4.6 – **Effects of a density bitmap**. The bitmap in (a) is reflected in the density pattern in (b).

A_{max}—which were invariant, previously—must be modified to account for the specified density. First, an axis-aligned bounding box is created around the given polygon, and projected onto the greyscale bitmap. The average of the greyscale values within this bounding box is then calculated, and used as an inverse weight on A_{min} and A_{max}—polygons mapped onto a lighter area in the bitmap will be compared against *smaller* threshold values, and will therefore be subdivided to a greater extent. Meanwhile, polygons mapped to darker areas will be compared against larger values, and will consequently tend to be subdivided less. As a result of these modifications, the `ItSub` algorithm can be easily parametrised to exhibit a significant variation of road density, as illustrated in Figure 4.6.

This also addresses a valid concern: that is, whether any given road-generation algorithm should be based on an initial enclosing polygon—most cities, in fact, have very vague boundaries, if any. The `ItSub` algorithm currently does require the starting polygon P; however, the city boundary can be "softened" through the use of an appropriate density bitmap. By gradually dropping the density towards zero at the city limits, the resulting road patterns can be made sparse enough to become nearly negligible.

Road Patterns

As with the variation of road density throughout a city, it is also common to find a variety of road patterns as described in Subsection 4.3.2: for example, the regular, grid-like pattern of many modern road networks is often disrupted by local terrain conditions or adaptations to legacy roadways. As above, a greyscale bitmap is used to represent the desired influence of a given road pattern (Manhattan or Arbitrary) in a particular area, with lighter areas indicating a strong influence, and darker areas a weaker one. In the case of road patterns, however, the component of interest is the branching angle of a new road, as opposed to the whole area of a polygon to be subdivided as in the case of changing road densities. Therefore, only a single point (the branching point of the new road) is mapped to the underlying bitmap, as opposed to an entire polygon. Greyscale values in the vicinity of the branching point are sampled, and the average value is determined; this value then corresponds to the influence of a particular road pattern on the branching angle at that point.



(a) "Abitrary" influence (b) Resulting subdivision

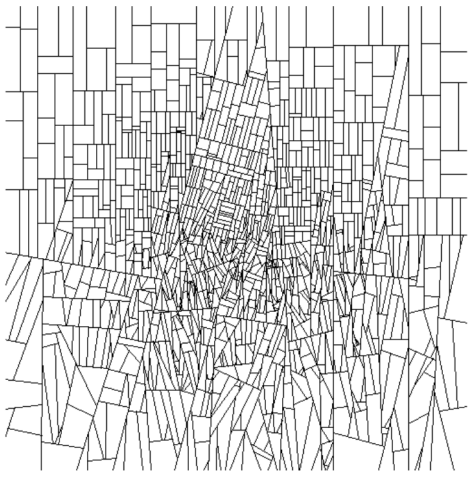
Figure 4.7 – **User-controlled road plan variations**. The bitmap in (a) defines a region of Arbitrary-influenced roads in the top-right corner of the city; (b) shows the resulting subdivision. Note how the roads seamlessly vary from being very grid-like and regular, to being much more chaotic, as desired.

Importantly, it must be noted that there are two bitmaps to sample in the current

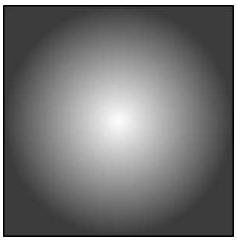
`ItSub` implementation, one for each of the two types of road pattern: Manhattan, and Arbitrary. The calculation described above is therefore carried out on both bitmaps, and the resulting values compared; the pattern with the greater influence is then reflected in the choice of μ and σ for the perturbation Gaussian (see 4.3.2), which then determines the appearance of the resulting subdivisions. An example of this use of bitmaps to influence the appearance of road patterns is shown in Figure 4.7.

Combining Inputs

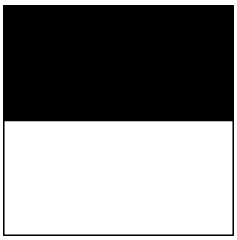
Interestingly, the parametrisations of both road density and patterns are mutually independent; the former only affects the total size of resulting polygons, while the latter changes the angle that roads make with each other. Since neither operation affects the other, they can be integrated simultaneously into the general `ItSub` algorithm. An example of the use of both types of input simultaneously is shown in Figure 4.8.



(a) A completed subdivision



(b) Density map



(c) Arbitrary influence map

Figure 4.8 – **Density and road plan influence maps used simultaneously**. The road map in (a) seamlessly integrates both the density map (b) and the Arbitrary influence (c).

4.3.5 Blocking Polygons

As outlined in Chapter 1, one of the goals of the Arda system and the current project in general is to enable artists to quickly integrate their own work with other, procedurally-generated content. Another additional capability is therefore deemed useful: to allow users to specify regions within the original polygon which will specifically *not* be subdivided. We call these regions *blocking polygons*, and their use will enable artists to create their own specific buildings or structures and reserve a space within the resulting cities in which to insert their models.

There are two changes to be made to the `ItSub` algorithm to accommodate blocking polygons. First, the algorithm must accept, as a new input, a set of polygons that will be used as blocking polygons. We call this set $S_{blocking}$, and note that it can potentially be empty (i.e., no blocking polygons); furthermore, we assume that the blocking polygons provided are all non-overlapping, and contained by the original `ItSub` polygon, P.

The second modification takes place when a bisecting line (L_{bisect}) has been successfully chosen. Before determining if the resulting polygons are valid or "acceptable," L_{bisect} is compared against all polygons in $S_{blocking}$—the current `ItSub` implementation uses the JTS computational geometry suite [27]. If the bisector does *not* cross through any blocking polygons, then the working polygon is subdivided by L_{bisect} as in the original implementation: sub-polygons p_1 and p_2 are generated and independently evaluated. However, if the bisector *does* pass through one or more blocking polygons, a number of intermediate steps are required. First, a set S_i is created of all blocking polygons that do intersect with L_{bisect}. Every polygon within S_i is then subtracted from the working polygon, resulting in a working polygon with "holes." Next, L_{bisect} is also subtracted from this working polygon; this results in two polygons, $p_1^{'}$ and $p_2^{'}$, which are similar to p_1 and p_2, but which follow the contours of the blocking polygons. These are then evaluated independently as before. Since the space within the blocking polygon is added to neither $S_{oversized}$ or $S_{parcels}$, we ensure that it is never subdivided, as required. This process is fully illustrated in Figure 4.9.

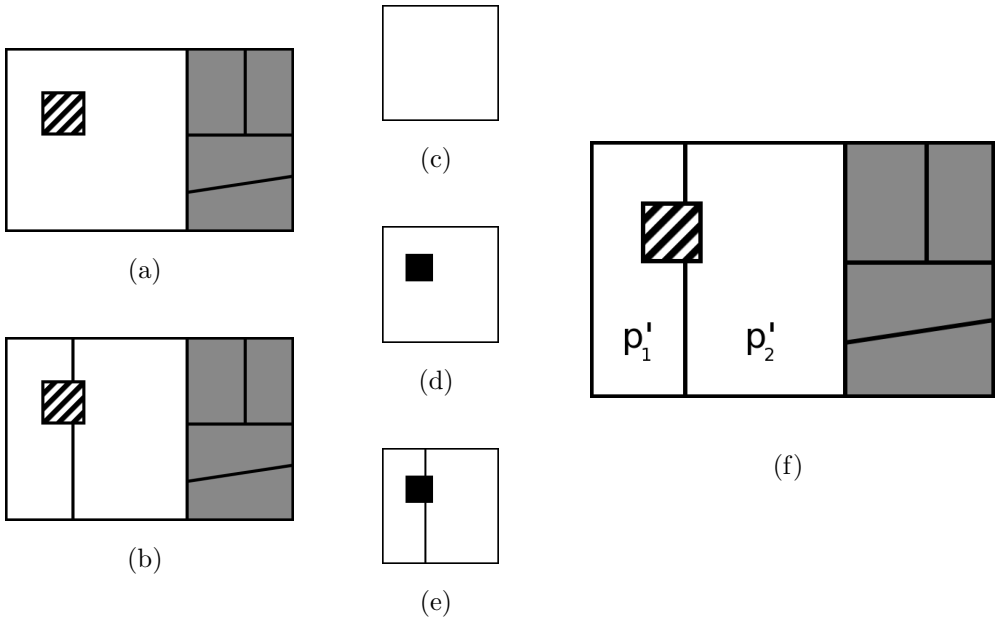


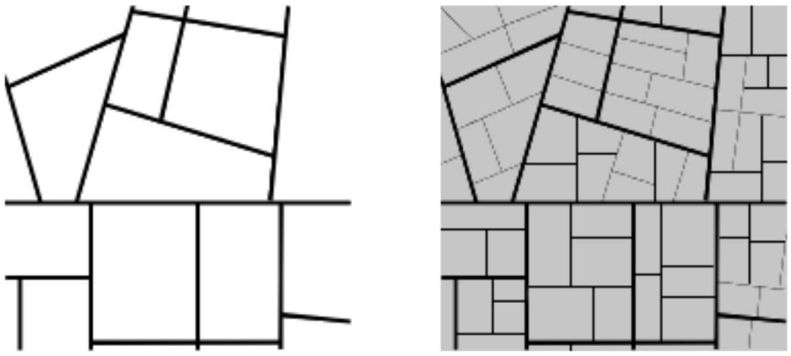
Figure 4.9 – **Blocking polygons**. (a) Completed polygons (grey), current working polygon (white), and blocking polygon (striped). (b) Randomly-chosen L_{bisect}. (c) Working polygon. (d) Working polygon with all intersected blocking polygons subtracted. (e) Working polygon with blocking polygon and L_{bisect} removed. (f) Final subdivision of working polygon, with two new polygons $p_1^{'}$ and $p_2^{'}$ following contours of the blocking polygon.

4.4 Improved Algorithm

The multitude of changes in the previous section prompts us to re-define our original `ItSub` algorithm to account for all modifications. The improved version of `ItSub` is shown in Listing 2 (see page 59), and contains all modifications listed above. In Chapter 5, we examine the output of `ItSub` under a variety of initial conditions and evaluate it from a qualitative perspective.

4.5 Allotment Subdivision

We have thus far discussed the `ItSub` algorithm purely in the context of generating urban road networks. However, an examination of Table 3.1 suggests another use for



(a) (b)

Figure 4.10 – **Using `ItSub` for allotment generation**. (a) shows the resulting road map created by `ItSub`. (b) shows the result of applying `ItSub` to each land parcel from the previous step.

the technique. After the road network module has completed in a particular city, the latter contains a number of polygons that represent land parcels; these must then be divided randomly in order to create the varied allotments required for individual buildings. The `ItSub` algorithm is ideally designed for this task, being by definition intended to subdivide polygons.

To create allotment subdivisions, `ItSub` proceeds in much the same way as for road networks, as described above. Each parcel in the city—representing "vacant land"—is used as an input for `ItSub`; that is, the parcels are the starting polygons (P) which must be subdivided. Note that these polygons may have been generated by `ItSub` at the road network generation level, or by any equivalent algorithm. Finally, the `ItSub` algorithm is applied to each land parcel polygon in turn. Whereas in road network generation, the algorithm returned a set of parcels, it now returns a set of *allotments*, which completes the definition of the given city being generated.

Importantly, the allotment polygons generated here (and their contained buildings) have the additional requirement that they are typically expected to be accessible from the road itself; the Arda Framework and pipeline ensure that each polygon has a road-facing edge regardless of the road generation algorithm chosen. In the case of

`ItSub`, this is currently ensured by careful setting of the A_{min} and A_{max} parameters; if the allotments are no smaller than a given size, then the resulting allotment subdivision will not be small enough to accommodate interior polygons, and each allotment will therefore have road access. This is, however, a weak guarantee, and future work would include a more rigourous assurance of road access.

Note also that some assumptions can be made regarding the `ItSub` algorithm in this context. Since the majority of real-world allotments exhibit some regularity (i.e., are bound by right angles), the use of "road patterns" as a defining characteristic no longer applies; the algorithm can be simplified to assume a global "Manhattan-style" pattern—that is, the resulting allotments are consistently grid-like, as expected. Furthermore, we assume in the current implementation that there is no density map affecting allotment sizes—it can be considered constant across P (the parcel). These alterations produce allotments as shown in Figure 4.10.

4.6 Integration into Arda

Having developed the `ItSub` algorithm, we can now begin to integrate it within the Arda system presented in Chapter 3. We note that the Arda Framework has three distinct types of content—Terrain, Cities, and Buildings (see 3.3.1)—and five "interfaces" into which content-generating algorithms can be inserted (Table 3.1). As described at the outset of this chapter, the `ItSub` algorithm is intended to replace previous, conceptually-complex road-generation algorithms; we must therefore create a generation module to wrap around `ItSub`, provide the appropriate inputs, and return the acceptable type of content to the Arda pipeline. Furthermore, we must create a parametrisation module to be used by the Arda Toolkit in presenting algorithm parameters to the user.

4.6.1 Generation Module

In Subsection 3.3.3, we described the principal components of a generation module: input (from higher-level content in the Arda hierarchy, and from external parameters),

the algorithm, and output.

Given that we are creating a Road Network generation module, we know which higher-level modules have already been executed, and thus which content is available for querying: when the `ItSub` generation module is executed, it is guaranteed that the underlying terrain has been generated, and that all cities have been placed in the virtual world (i.e., their positions and radii have been determined); a specific `ItSub` module will operate on a specific city, whose roads and parcels have not yet been defined. While the `ItSub` algorithm does not yet adapt to terrain variation, this is would be a desirable extension, and is discussed in Chapter 6. On the other hand, city radius can be used to determine a starting polygon for the algorithm; for instance, a circular or polygonal shape can be created with the given radius around the city's centre. Another option would be to allow the user to supply a starting polygon, along with all other parameters, as in the following subsection.

The algorithm implemented by the `ItSub` generation module would simply be same modified algorithm defined in Listing 2, with all parameters (starting polygon P, A_{min}, A_{max}, and $S_{blocking}$) provided by the user. Finally, the `ItSub` module has the responsibility to provide a set of road segments and a set of land parcels to the city. These are precisely the outputs of the `ItSub` algorithm: S_{roads} and S_{parcel}, respectively. A similar generation module can be created to fill the role of allotment subdivision.

The `ItSub` algorithm is therefore easily tailored to the general requirements of all generation modules, and can be very rapidly integrated into the Arda Framework for use in content generation.

4.6.2 Parametrisation Modules

The parametrisation of a given iteration of the `ItSub` algorithm is relatively straightforward. For the generation of road networks, the following parameters are made available to the user for modification (default values are shown in brackets):

- Polygon P—the starting polygon of the `ItSub` algorithm (Default: assume the city spans the entire width and height of the underlying terrain).

- A_{min}, A_{max}—the threshold values used within `ItSub` (Default: depends upon the scale of the original terrain).
- Density map—a bitmap showing the desired density of roads within the resulting subdivision (Default: none).
- Manhattan, Arbitrary maps—bitmaps showing the desired variation of road patterns within the resulting subdivision (Default: none).
- $S_{blocking}$—a file containing definitions of polygons not to be subdivided by `ItSub` (Default: none).

Similarly, a parametrisation module is required in order to use `ItSub` to generate allotment subdivisions; however, as outlined in 4.5, a number of simplifying assumptions can be made in order to eliminate several of the parameters. All that is required in this case is a means of changing the threshold values A_{min} and A_{max}, which are assumed to be smaller than the threshold parameters used in road network generation. With these values defined, `ItSub` can properly subdivide the provided parcels.

The `ItSub` algorithm is therefore quite flexible: with the appropriate generation module wrapper, it can be integrated seamlessly into the Arda Framework and content-generation pipeline; and with the appropriate parametrisation module, it can be easily controlled by the user through an external application.

4.7 Summary

In this chapter, we presented *Iterated Subdivision*, or `ItSub`—an algorithm for the repeated, controlled subdivision of polygons. The algorithm can be used, as shown above, for the procedural creation of complex urban road networks, and compares favourably to other such algorithms in terms of conceptual simplicity and ease of parametrisation. The algorithm has several means to ensure an acceptable aesthetic quality; furthermore, it can generate more complex road patterns through the use of input bitmaps to control localised density of the roads, as well as the type of road pattern exhibited (Manhattan or Arbitrary). In addition, it is not limited to road

generation, but can be re-used for other procedural content generation tasks. Further potential improvements are outlined in Chapter 6.

The `ItSub` algorithm is shown to be easily integrated into the Arda system described in Chapter 3. In the next chapter, we examine the types of road networks and allotments generated by the algorithm, and experiment both with `ItSub` and the Arda system as a whole, in order to evaluate its use in generating video game content.

Algorithm 2: Improved Iterated Subdivision

Input: Polygon P, A_{min}, A_{max}, R_{max}, D_{min}, $S_{blocking}$

Output: S_{parcel}, S_{roads}

$S_{oversized} \leftarrow P$

$S_{parcel} \leftarrow S_{roads} \leftarrow \emptyset$

repeat

  $P_{working} \leftarrow S_{oversized}.poll()$

  $\{A'_{min}, A'_{max}\} \leftarrow \{A_{min}, A_{max}\}$ modified by density map

  repeat create random bisector

    $L_{bisect} \leftarrow$ random bisector, angle based on road plan inputs

    snap L_{bisect} endpoints if necessary (see D_{min})

    if $L_{bisect} \cap S_{blocking} \neq \emptyset$ **then** `/* intersects blocking polygons? */`

      $\{p_1, p_2\} \leftarrow$ subdivision with blocking polygons subtracted

    else

      $\{p_1, p_2\} \leftarrow$ regular subdivision

    end

  until $|p_1| \geq A'_{min}$ **and** $|p_2| \geq A'_{min}$ **and** $\frac{diam}{width} \leq R_{max}$

  $S_{roads} \leftarrow S_{roads} \cup L_{bisect}$

  foreach *polygon* p_i *in* $\{p_1, p_2\}$ **do**

    if $p_i.area > A'_{max}$ **then**

      $S_{oversized} \leftarrow S_{oversized} \cup p_i$

    else

      $S_{parcel} \leftarrow S_{parcel} \cup p_i$

    end

  end

until $|S_{oversized}| == 0$

Chapter 5
Evaluation

The previous chapters have introduced two major concepts: the Arda system for procedural video game content creation; and the Iterated Subdivision algorithm for rapid, parametrised creation of urban road networks (and, as an extension, for allotment subdivision). The next step is to evaluate their respective performance in light of the goals with which we originally set out. We begin in Section 5.1 by evaluating `ItSub` in terms of raw performance—the time taken to subdivide large polygons—as well as its realism in comparison to actual cities. Next, in Section 5.2 we examine the benefits of parallelisation within the Arda Framework, as it is designed to generate many different content elements simultaneously. Finally, in Section 5.3, we evaluate the use of the Arda system as a whole in generating content for an existing video game project.

All experiments in this chapter are performed on a computer with an AMD Phenom 9500 2.2GHz quad-core CPU and 4GB of RAM, running Xubuntu 8.10 (Intrepid Ibex).

5.1 Testing `ItSub`

The `ItSub` algorithm provides a number of *qualitative* advantages over other road network generation techniques, as discussed in Chapter 4. However, we wish to

perform more *quantitative* analyses of the algorithm in order to evaluate its overall characteristics.

5.1.1 Performance

A central performance concern is how `ItSub` scales in terms of generated city size. Several substeps repeat based on randomization, and it is important to determine whether this has practical impact. To evaluate scalability the size of the original polygon is progressively increased, while maintaining the same minimum subdivision size. Note that for the purposes of even large computer games the most important size range is much smaller than that of any large existing, real-life city.

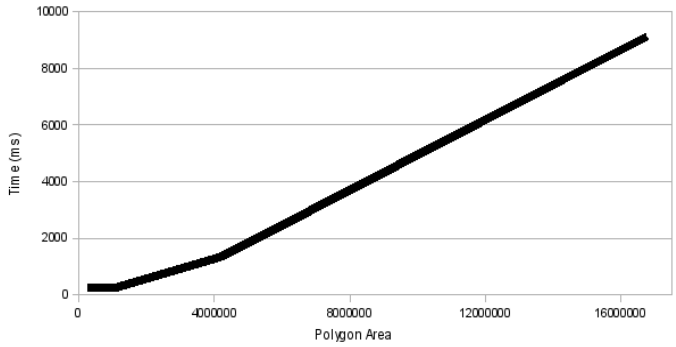
		No bitmaps		**All bitmaps**	
Map Length	**Map Area**	**# Roads**	**Time (ms)**	**# Roads**	**Time (ms)**
512	262144	203	236	201	1059
1024	1048576	812	253	802	4502
2048	4194304	3258	1339	4030	26096
3072	9437184	7338	4602	7239	61276
4096	16777216	13037	9138	12851	126459

Table 5.1 – Influence of starting polygon area on `ItSub` running time.

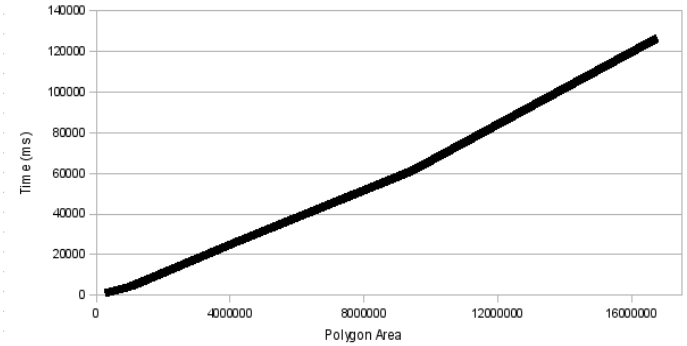
To perform these experiments, we define square-shaped initial polygons of varying size, starting with 512 pixels on a side and gradually increasing. `ItSub` is then executed on each polygon, with A_{min} and A_{max} being held constant, to ensure roughly the same amount of subdivisions per unit square (in these experiments, we set $A_{min} = 64.0$ and $A_{max} = 2048.0$ for testing purposes). Each input polygon is tested 10 times under these conditions, and the average number of roads created and average running time is then calculated. Finally, to evaluate the effect of using parametrisation bitmaps (i.e., density map, Manhattan and Arbitrary distribution maps), we perform one such series of tests without the maps, and another using all three.

The results of the performance tests are summarised in Table 5.1, and shown graphically in Figure 5.1, with the caveat that memory constraints forced the test

to halt at starting polygons of 4096 pixels per side. As the area to be subdivided becomes larger, the running time for the `ItSub` algorithm increases approximately linearly. The use of bitmap parametrisation adds significantly to the cost, largely due to the use of inefficient, generic routines for accessing bitmap content in the prototype design, but maintains the linear relation.



(a) No Bitmaps



(b) All Bitmaps

Figure 5.1 – **Performance of `ItSub` when varying starting polygon area**. (a) shows the performance of `ItSub` when no bitmaps are present; (b) shows the performance using all bitmaps (Density map, Manhattan and Arbitrary maps).

Of course the numbers used in this evaluation (that is, number of road segments generated) require some kind of baseline. For instance, the fact that `ItSub` is capable

of creating 13,037 roads in 9 seconds is meaningless without relating it to a real-world example. Dupuis and Chopard provide such a sense of scale with their simulations of traffic in Geneva, Switzerland. In modeling this city, they create a road network of discrete road junctions and segments and note that "[the] full network comprises 3145 road segments and 1066 junctions" [4]. Meanwhile, the city of Geneva itself occupies $15.86km^2$ and is home to roughly 185,500 inhabitants [31]. Coupled with the known number of road segments and the information in Table 5.1, it can therefore be said that the `ItSub` algorithm is capable of generating a city roughly the size of Geneva in 1.3 seconds if no parametrisation bitmaps are used, or roughly 25 seconds if the bitmaps *are* used. We also note that New York City—the canonical city used when discussing road generation algorithms—occupies an area of $1{,}214km^2$ and is home to 8.2 million inhabitants. Though a direct formula for calculating the required number of road segments from this data is currently unknown, we can perform a rough calculation based on the respective areas of both regions: New York City is 76 times larger than Geneva, so we might expect an equal proportion of additional roads. This gives $3145 \times 76 = 239,020$ roads in a potential application of `ItSub` to New York City; referring to Table 5.1, we can deduce that such a network would take roughly 2 minutes, 47 seconds to complete without parametric bitmaps, or 39 minutes and 12 seconds with all bitmaps. While the latter figure may seem large in comparison, it is nevertheless suitable for off-line generation, particularly given the real-world size of the city being generated. Finally, it must be emphasised that these values—the number of roads in Geneva, the proportionality between city size and number of road segments, and so on—are extremely approximate. The numbers reported by Dupuis and Chopard are estimated, and no exact formula is known to relate numbers of roads with size, density, or population of a given city. However, these numbers may give at least a rough sense of the speed with which the `ItSub` algorithm can recreate complex, "life-sized" cities.

5.1.2 Realism

While the analysis above gives some idea of the time required by `ItSub` to generate roughly life-sized cities, it says nothing about the inherent realism of the result. Unfortunately, few metrics exist to evaluate urban road networks in this way. The design and analysis of roads does appear in academia, but the primary concern in these cases is efficiency/throughput and safety of roadway users—see for example, the American Association of State Highway and Transportation Officials' "Green Book" [1]. In the next chapter, we discuss the need for more analytical metrics of cities and road networks that can then be used to evaluate individual road-generation techniques, or perform comparisons between several.

In the absence of definite metrics, analyses of `ItSub` (or any algorithm for creating road networks) must necessarily be of a qualitative nature only. In Figure 5.3 (see page 69), we demonstrate the use of `ItSub` on real-world landforms: the islands of Manhattan and Montreal. The starting polygons (P) for both islands were generated by hand, and `ItSub` executed using the density and road pattern maps shown. The map of Manhattan consists of roughly 2700 road segments having taken approximately 7.5 seconds to be generated. In addition, a blocking polygon was used to keep the mass of Central Park (towards the right of the image) free of subdivisions (i.e., roads). Meanwhile, the Montreal road map contains approximately 10000 road segments, and ran for 70 seconds—roads are visibly much more densely distributed than in 5.3(b). In combination with the given parametric bitmaps, `ItSub` thus creates qualitatively convincing road networks adapted to the real-life cities of Manhattan and Montreal. While the resulting networks are perhaps not as extensive as their real-life equivalents, the speed with which they generated shows the practicality of `ItSub` in creating large cityscapes quickly.

5.2 Parallelisation of Arda Framework

One of the primary goals when designing the Arda system was to enable a certain degree of parallelism. For example, a building being generated at one end of a city

has no influence on the generation of another building at the other end, and thus generating them in parallel runs no risk of conflict while reducing the overall run time. In Chapter 3, we saw how the content generation pipeline of the Arda Framework is designed to allow such parallel computations; we now perform tests in order to evaluate these benefits.

Tests are performed by generating full content trees incorporating the three content types defined in Chapter 3—Terrain, City, and Building. Content generation proceeds as follows. First, a single Terrain is generated, and several sites in the resulting map are chosen for generating Cities. Next, a new thread is created for each City to be generated, and (in parallel) every City creates its own local road network and allotment subdivisions, or Buildings. Terrain is generated as a flat map—since there is only ever one (non-parallelisable) Terrain, this step is negligible and kept as simple as possible. City placement is random, with all Cities being given the same overall size. Road networks are generated using the grid-based algorithm described in Chapter 3: each city is created as a square with 20 horizontal roads and 20 vertical—or in other words 21 buildings horizontally by 21 buildings vertically—resulting in a total of 441 buildings per city. Note that a constant number of buildings in each city renders the test much more stable; other randomised road-generation techniques (such as `ItSub`) would create a random number of buildings in every city, adding to the statistical noise of the content creation process. These tests, moreover, are intended to examine parallelization of the Arda system more than the specific behaviour of `ItSub`. Next, allotment subdivision actually performs no subdivision at all, simply re-labelling each land parcel as one large allotment. Finally, building façades are generated by simply offsetting the building polygon from the surrounding roads. See Chapter 3 for the implementation of these generation modules.

The Arda Toolkit is used to perform the generation listed above, with a variable number of Cities. For each given number of Cities, a virtual world is created 10 times, and the average time to complete the process is computed. This series of tests is first executed with parallelism activated (the default state of the Arda Framework), and then once again with parallelism turned off. In the latter case, a single control thread is responsible for generating the Terrain, each City, and each Building. We

Num. Cities	Num. Buildings	Single thread (ms)	Parallel (ms)	Speedup
1	441	592	283	2.09
2	882	1146	419	2.73
4	1764	2171	609	3.56
8	3528	4178	1093	3.82
16	7056	8131	2495	3.26
32	14112	16287	4476	3.64
64	28224	32409	12867	2.52

Table 5.2 – Influence of parallelisation on running time of Arda pipeline.

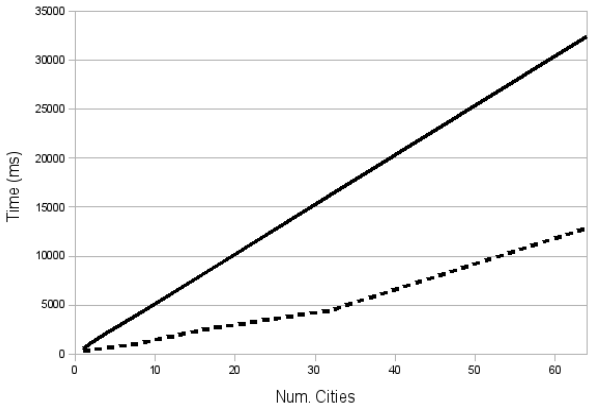


Figure 5.2 – **Performance of Arda pipeline with and without parallelisation**. The solid line represents the time taken to create content with a single execution thread; the dotted line represents the same, but using a separate thread for each data element.

therefore expect that the single-threaded tests should take appreciably longer than the multi-threaded ones, since content elements are created one at a time. Finally, we note once more that all tests are performed on a quad-core system, whereas a full parallelisation analysis would normally involve significantly varying the number of available cores as well—from a single CPU, up to dozens. The resulting analysis here is therefore less detailed than others, but should serve to adequately demonstrate the value of parallelisation within the Arda Framework.

The results of the parallelisation tests are summarised in Table 5.2, and shown in

Figure 5.2. It is clear, by examining this data, that the parallelisation of the Arda content generation pipeline results in a significant savings in terms of execution time, as predicted. The speedup observed in each sample above averages around 2.5 times, with a maximum of 3.82—close to the theoretical speedup of 4x that would be expected on a quad-core system. Finally, we note the relative simplicity of the data generated in these trials: a flat terrain, a limited number of cities, and very simple buildings. Savings will become even more important as the data being generated becomes more complex, and as the corresponding pipeline hierarchy branches exponentially; furthermore, deploying the Arda Framework on much larger distributed systems would lead to even greater performance.

5.3 Integrating Arda into Mammoth

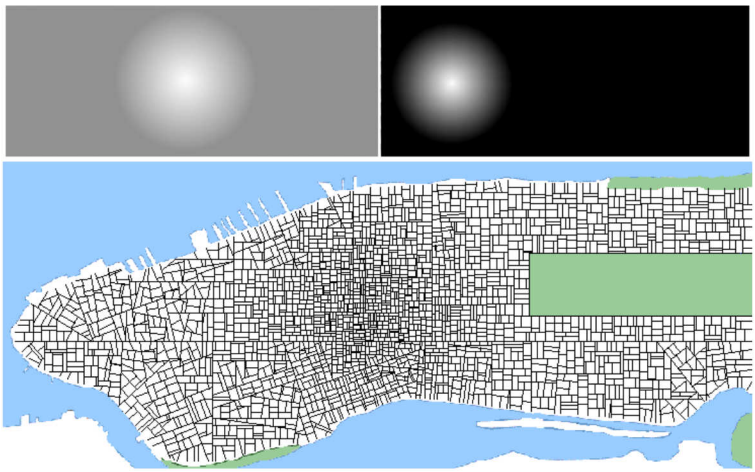
In Chapter 1, we discussed the growing need for tools capable of creating vast virtual worlds quickly and procedurally, and expressed an interest in making such a tool modular and extensible, so as to accommodate the plurality and variety of content-generation algorithms. We now look at the process of integrating our Arda system into an extant video game project, and using it to create much larger virtual environments than was previously possible.

An ongoing project in the School of Computer Science at McGill University has been the development of *Mammoth*, "a massively multiplayer game research framework" allowing students, professors, and researchers to experiment with various techniques in AI, distributed computing, databases, and so on, within the constraints of a massively multiplayer online (MMO) game [10]. Being a large game project potentially able to make use of extensive virtual environments, Mammoth is an ideal testbed for the type of content generated by the Arda system. The integration of Arda-generated content into Mammoth has therefore been complementary to the process of developing Arda itself.

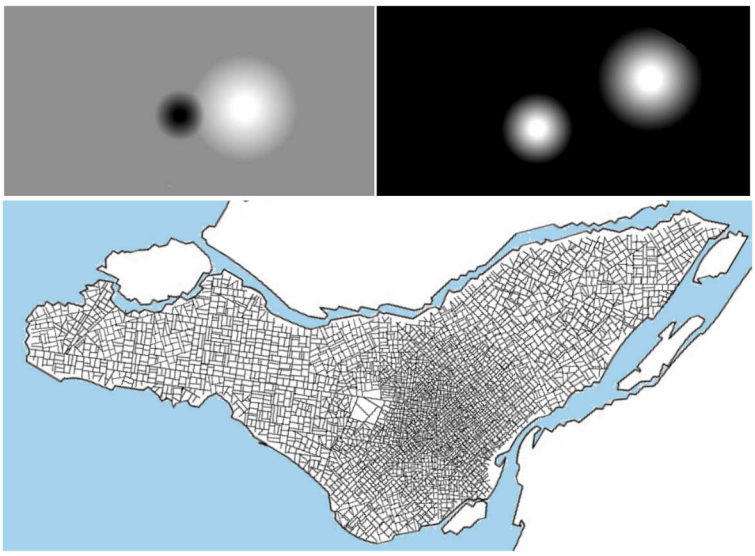
Adapting the Arda system to a developing project such as Mammoth is complicated by the latter's constantly changing specifications. Mammoth underwent a

drastic shift from 2D to 3D environments during the course of Arda's development, which emphasised the importance of properly defining static content types in the Arda Framework implementation (Terrain, City, Building—see Chapter 3). As high-level design decisions change the use and capability of data, these content types must be re-evaluated in order to ensure consistency. Here, the abstractness of the chosen data representations enabled a smooth transition between various Mammoth revisions. Changes in Mammoth's design required fundamental alterations to the game engine, a task which lasted several weeks; conversely, the Arda static content types were flexible enough to require only minimal changes over the course of several days—changes for the most part limited to the process of exporting data to the appropriate format, not in the actual definition of data itself. Properly designing the Arda static content types to be suitably abstract is thus important to ensure the greatest possible flexibility in the Arda system.

Once the general design of Mammoth became more stable, generating the content itself proved comparatively simple. The Arda Framework and Toolkit had already been fully designed; the Toolkit was simply modified to assemble the content created by the underlying Framework, and export this data to the appropriate formats for the JMonkey graphics engine used by Mammoth [20]. Results are shown in Figures 5.4 and 5.5 (see pages 70 and 71). As a rough guide, the content shown in these images took less than one minute to generate from the moment the Arda Toolkit was opened. While this content is not yet on par with contemporary open-world video games, the size of the generated worlds, the speed with which they were created, and the potential of more sophisticated content-generation algorithms should emphasise the possibilities of procedural content generation.

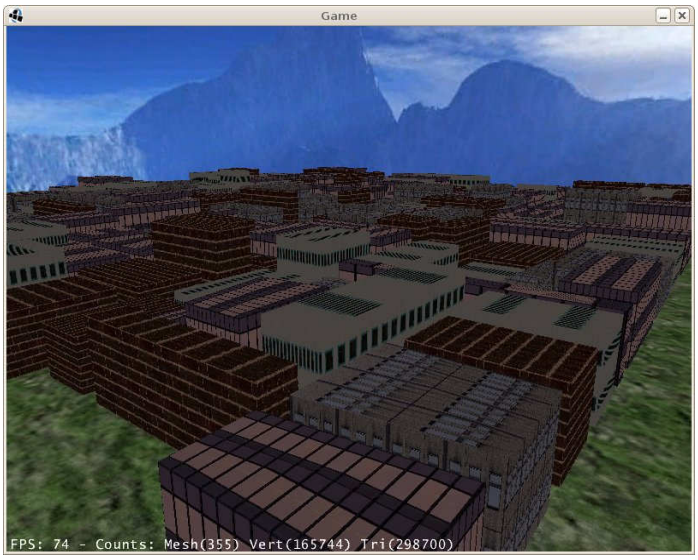


(a) Manhattan



(b) Montreal

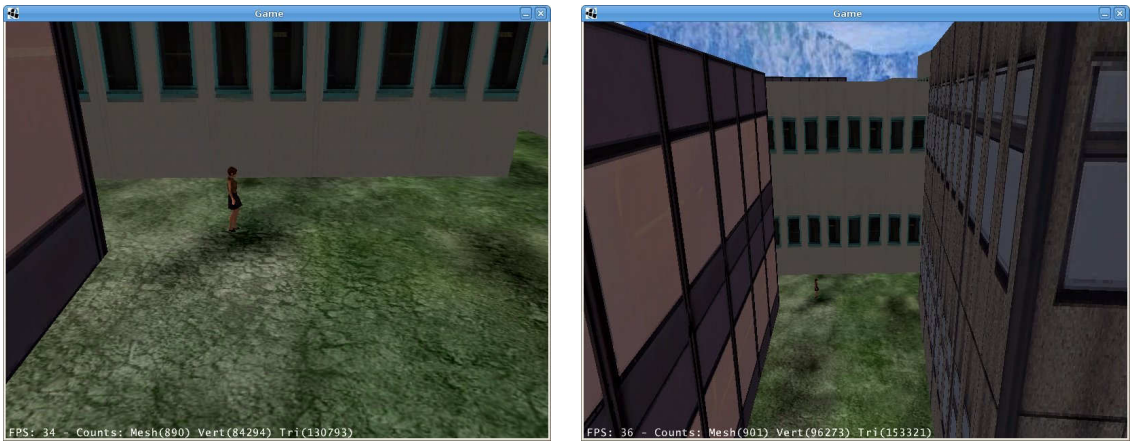
Figure 5.3 – `ItSub` **applied to Manhattan and Montreal islands**. Each figure shows a density map (top-left) and an Arbitrary road map (top-right), with the resulting subdivision below. Note the varying density and chaotic branching angles corresponding to the input bitmaps, as well as the use in (a) of a blocking polygon on the right of the image.



(a)

(b)

Figure 5.4 – Screenshots of Arda-generated content in Mammoth



(a) (b)

(c)

Figure 5.5 – A sense of scale in Arda-generated content

Chapter 6
Conclusion & Future Work

6.1 Conclusion

In Chapter 1, we emphasised the increasing need for procedural content generation techniques in video games. Traditionally, all art assets in a game are constructed manually by artists using a variety of tools; however, this production model rapidly becomes prohibitively expensive—in both time and money—as data storage and graphics capability become increasingly powerful. At the same time, procedural content generation techniques are numerous and varied, prompting the question as to why more games do not make use of this type of automatic content creation. This led us to propose the Arda system for procedural content generation, a system by which any number of varied algorithms can be interchanged, and by which content can then be created and exported to a particular game's format.

Chapter 3 saw the development of the Arda system itself, divided between two components: the Arda Framework, and the Arda Toolkit. The Arda Framework is an internal representation responsible for defining content, generating it in the proper order, and managing the different content-generating algorithms used. The Framework uses specific data representations to ensure that all algorithms have the same "view" of data; the current implementation of the Arda Framework defines Terrain, City, and Building content elements, and all algorithms must conform to these structures. Next, the Framework has the capability to generate content in a "top-down" manner:

large structures (e.g., Terrain) are generated first, followed by elements of gradually decreasing scale (e.g., Cities, then Buildings, and so on). The strict hierarchy guarantees that content at lower levels can refer to higher-level content as parametric inputs. Furthermore, the generation can be performed in parallel, allowing a drastic reduction in run-time when generating vast amounts of non-overlapping content. Finally, the Arda Framework provides a well-defined API enabling programmers to add new algorithms to the system. Higher-level content and externally-controlled parameters are provided as inputs to so-called "generation modules," which wrap around a content-generating algorithm and return the generated content in a format appropriate for the content elements defined previously. These generation modules are then managed by the Framework, and can be selected and swapped by the user at run-time.

The Arda Toolkit is a graphical user interface (GUI) that allows users to interact more easily with the Arda Framework, which is internal to the Toolkit itself. The Toolkit displays, for the user, the hierarchy of content elements created by the Framework; the user can then select nodes and determine which content-generating algorithms are to be used at every point during the generation process. Each algorithm provided to the Framework also has a corresponding parametrisation module provided to the Toolkit; this module presents algorithm-specific parameters to the user, and allows the latter to affect the algorithm as it operates in the Framework. Once the user is satisfied with the selection and parametrisation of algorithms, the content is generated entirely algorithmically, and can be saved by the user in an appropriate format. Several basic algorithms and associated parametrisation modules have been created for use in the Framework and Toolkit, respectively, and ensure that a new user can immediately start creating content without any programming required.

While the basic modules provided for the Arda Toolkit are functional, they are lacking in realism or complexity; in order to improve the results of content creation, more powerful algorithms are required. A number of techniques exist for generating urban road networks, for instance, but these are either overly complex or difficult

to parametrise. Therefore, in Chapter 4, we developed the Iterated Subdivision algorithm, or `ItSub`, to allow rapid, customisable road network generation in a much more straightforward way. The algorithm repeatedly subdivides polygons until they are within a certain size threshold; the resulting bisecting lines are interpreted then as road segments, and the spaces between them as "parcels" of land. The algorithm can be parametrised through the use of greyscale bitmaps representing the varying density of the given city, or the change of road pattern between Manhattan (grid-like branching points) and Arbitrary (branching angles of any size) styles; furthermore, the user can specify regions within the original polygon that are not to be subdivided, which can therefore represent obstacles or regions within which manually-generated content will be inserted. The completed `ItSub` algorithm thus provides a conceptually simple, easily parametrised technique for generating arbitrary but realistic road patterns, and can even be re-used (with fairly trivial modifications) to subdivide resulting land parcels into individual allotments for buildings, another required module in the Arda content-generation pipeline.

In Chapter 5, we evaluated both the `ItSub` algorithm and the Arda system. `ItSub` was shown to perform well as the size of the original polygon to be subdivided is increased. Furthermore, we saw that the available parametrisations provide adequate control over the algorithm, which is therefore capable of creating a wide variety of road networks very quickly, and qualitatively on par with other, more complex algorithms.

Finally, the Arda content generation system was adapted in order to produce content for a video game currently under development at McGill University. Though the game was in a state of constant development—emphasising the need for accurate data representations in the Arda Framework—it was possible to test large environments successfully within the game. The experience of producing procedural content for the game was straightforward and rapid, and newly-generated worlds were playable within a matter of minutes.

The traditional approach to content generation for vast environments—hiring more artists and spending more time on the process—is rapidly approaching a limit in terms of feasibility and accessibility. The logical solution is the introduction of the Arda system for procedural content generation, or systems like it; using them,

game developers can more easily take a hybrid approach to content creation, where both procedural and manually-created environmental assets are merged seamlessly, creating massive and unique video game worlds.

6.2 Future Work

While the Arda system and the `ItSub` algorithm are both well-developed in this thesis, a number of improvements can be suggested for either one to be extended in the future.

6.2.1 Improving the Arda Framework

We have shown that the Arda content generation system can accomodate multiple, interchangeable algorithms to create content for an extant video game project. Further improvements aim to make the system much more flexible in order to create assets for a much wider variety of games.

Extend the Framework

Currently, only Terrain, Cities, and Buildings are represented as static data objects in the Arda Framework. However, the majority of open-world game environments contain many more types of content, such as rivers, bodies of water, vegetation patterns, individual plants, dungeons or landmarks, and so on. Adding these various types of content to the Arda Framework would open up the use of an even greater variety of algorithms, and lead to even richer virtual worlds as a result.

Increased Modularity

While the Framework can be extended as outlined above, in the end the choice of which content elements must be represented internally, and how these should be structured, is very game-specific. To make the Arda content generation system as flexible and as general-purpose as possible, it may be desirable in the future to allow

even data *types* to be modified or interchanged by the user, as opposed to simply the algorithms used to generate that content, as now. In such a system, programmers could define explicitly how to represent data (for instance, giving their own definition to the Terrain, City, and Building types); generation modules (and the algorithms around which they wrap) would therefore be forced to provide output specific to *these* specifications when executing. The Framework would serve a similar purpose as before: once a data hierarchy has been defined and the programmer has identified points where content generation (i.e., generation modules) are required, the Framework would then be delegated responsibility for generating the tree-like hierarchy of content—once again, this could still be level by level, and in parallel across branches.

An alternative would be to create a fully customisable API: custom data types and generation algorithms could be arranged in "building-block" patterns while a higher-level Framework monitors the correct sequence of content generation. This would address problems discussed in 3.3.2, though it would require a significant rewriting of the code, while the arbitrariness of structure would likely make parallelisation impractical or impossible.

New Types of Environment

Whether the Arda Framework is simply extended through the addition of specific new content types, or whether it is given the ability to use modular types as defined above, it would be desirable to see a variety of different content created using the system. Whether implicitly or explicitly, the discussion thus far has tended to suggest modern urban environments, such as those found in the *Grand Theft Auto* series, or fantasy environments, such as those in the *Elder Scrolls* series. The flexibility of the Arda content generation system could be further demonstrated through the generation of even more types of environments: Old West countrysides, space stations and future cities, Tron-like computer worlds, and so on.

6.2.2 Improving the Arda Toolkit

The Arda Toolkit is a preliminary attempt at GUI design. While it is functional, improvements can be suggested to make it more intuitive for an end user.

Improvements to the GUI

At present, all generation modules in the Framework and their associated parametrisation modules in the Toolkit are hard-coded. To fully realise the potential of the Arda system, the Toolkit should be modified to allow modules created externally to be imported and integrated immediately into the appropriate sub-systems. This requires that such modules are presented in a specific format (to facilitate importing), and would necessitate appropriate visual and functional changes within the GUI—for instance, providing a module management screen for the purpose. Other improvements would include: building towards a more suitable interface, as opposed to the file explorer-like window currently used; streamlining the process of regenerating multiple nodes simultaneously; and so on. These types of improvements will require extensive testing of the Toolkit and user surveys to determine areas requiring attention.

Testing for Quantitative Savings

Thus far, all analyses of the possible resource savings available through the use of the Arda system have been qualitative. However, while the potential of procedurally-generated content has been widely recognised, little work has been done to give this claim empirical weight. Ideally, a census of various video game companies would be performed to determine the allocation of a game's fiscal and temporal budget to the creation of game worlds; a comparison test could then be performed, using Arda to create worlds of similar size. While it is unlikely that a full and accurate report could be expected from most game companies, such a census would at least give an approximate indication of the benefits of using procedural content.

6.2.3 Improvements to `ItSub`

The Iterated Subdivision algorithm presented in Chapter 4 did fulfil the requirement for a conceptually-simple road network generation algorithm, as previously discussed. However, while several refinements were added to the algorithm, there are a number of desired improvements that have yet to be implemented.

Terrain Adaptability

Both the L-system and agent-based approaches to urban road network generation (among others) are capable of adapting to local terrain conditions; for instance, they might avoid generating roads with very large slopes, or tend to build denser patterns over large areas of level ground [8, 15, 17]. `ItSub` does not yet have this ability; however, adding terrain adaptability would increase the realism of the resulting subdivisions, and furthermore would likely incur only a small additional programming cost. Similar to the process of adapting `ItSub` subdivisions to greyscale density maps, one could imagine projecting a polygon onto a terrain height map; if the underlying terrain were too varied, subdivision could cease; or, conversely, subdivision could be encouraged further if the terrain were acceptably level. Another modification could examine the terrain beneath a bisecting line (L_{bisect}) for overly steep slopes, and discourage such roads from forming.

Automatic Initial Polygons

The `ItSub` algorithm requires a starting polygon, P, on which subdivision will be performed. Throughout this project, this polygon has either had a very simple form (i.e., a square) defined programmatically, or a more complex shape (such as the outline of the island of Manhattan) defined and input manually. In order to make the algorithm—and thus the Arda system into which it is integrated—more fully autonomous and more realistic, it is desirable to have starting polygons that, for instance, are defined based on the underlying terrain features, perhaps avoiding areas marked as "underwater" or following the outline of steep geographical features, for

instance. While this requires a level of pattern recognition and parametrisation beyond the scope of the current project, it would nevertheless improve the realism of the final subdivision.

Improved Blocking Polygons

The current use of blocking polygons in `ItSub` is effective in marking off areas that are not subdivided by the algorithm. However, degenerate cases exist, for instance when bisecting lines pass close to, but not touching, a blocking polygon, resulting in parcels or allotments too small for practical use. More careful scrutiny is therefore required before placing new bisecting lines. Additionally, an alternative method to achieve blocking polygons might also be tested: generating the full road network as before, and then simply removing the blocking polygons in a post-processing step. This might be computationally less demanding; however, we note that it would disallow the possibility of roads ending at the blocking polygon, as is the case presently, and might be desired.

Improved Context and Realism

Currently, `ItSub` produces a road map with only one "type" of (straight) road, as well as a parameter for becoming more or less grid-like. Realistic road patterns, however, require a greater range of expression and adaptation. Using population "sources" and "sinks," the graph-theoretical notion of a flow network might be used to determine major and minor roads, allowing road generation to proceed accordingly: large highways might have fewer connections, with smaller roads running in parallel, and so on. Curving roads are another desired feature, as they are evident in even the most grid-like cities (see Broadway on the island of Manhattan); likewise, dead ends could be added to road networks, or generated in post-processing. Roads could also be made to follow natural features such as coastlines or mountainous terrain, running parallel to these much like the highways mentioned above. Blocking polygons and/or sparsely-subdivided areas could be served or bounded by much larger roads. All of these features could contribute to an increased realism in the resulting road network.

New Parametrisation

The original `ItSub` algorithm was modified in Algorithm 2 to allow for bitmap parametrisation—greyscale images defining road density and branching angles. While this enabled a significant customisation of the resulting road plans, we can imagine other potential uses for such bitmaps, representing further kinds of information. For example, bitmaps could represent the distribution of "downtown" versus suburban areas; of older versus more modern neighbourhoods in the city; a distribution of income; and so on. These types of input would then affect road generation in the appropriate ways, and could perhaps even be inherited by lower-level algorithms: for instance, building façades could subsequently be parametrised by the relative age of the neighbourhood in which they are located.

Comparative Tests and Quantitative Metrics

The performance of the `ItSub` algorithm has been evaluated in Chapter 5, and it was determined to be acceptable for practical purposes. However, proper analysis requires that the technique be measured against other possible algorithms, such as L-systems and agent-based approaches. Measurements could include tests of pure performance—comparing the time required to generate road networks in cities of equivalent size, or comparing the time required per individual road segment, for instance—or tests of a more qualitative measure—evaluating the ease with which one algorithm can be used and controlled compared to another, coupled with an assessment of the resulting road networks.

Other tests and quantitative metrics could also be devised for use in determining how "city-like" the results of `ItSub` can be—that is, comparing the road networks created by `ItSub` with conditions in actual cities. Metrics such as road density, average road branching angle and variance, distribution of road lengths, and so on, would give weight to a claim that a particular road network generation algorithm (not simply `ItSub`) performs well in approximating real life cities. For the moment, however, no such metrics exist, and only qualitative arguments can be made.

Bibliography

[1] AASHTO. *A Policy on Geometric Design of Highways and Streets*. American Association of Highway and Transportation Officials, 2004.

[2] Bethesda Softworks. The Elder Scrolls: Daggerfall. Online, 2009. `http://www.elderscrolls.com/tenth_anniv/tenth_anniv-daggerfall.htm`, last visited Jan. 28, 2009.

[3] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286, New York, NY, USA, 1998. ACM.

[4] A. Dupuis and B. Chopard. Cellular Automata Simulations of Traffic: A Model for the City of Geneva. *Networks and Spatial Economics*, 3(1):9–21, January 2003.

[5] K. R. Glass, C. Morkel, and S. D. Bangay. Duplicating road patterns in south african informal settlements using procedural techniques. In *Afrigaph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 161–169, New York, NY, USA, 2006. ACM.

[6] S. Greuter, J. Parker, N. Stewart, and G. Leach. Real-time procedural generation of 'pseudo infinite' cities. In *GRAPHITE '03: Proceedings of the 1st international*

conference on Computer graphics and interactive techniques in Australasia and South East Asia, pages 87–ff, New York, NY, USA, 2003. ACM.

[7] E. Hahn, P. Bose, and A. Whitehead. Persistent realtime building interior generation. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 179–186, New York, NY, USA, 2006. ACM Press.

[8] T. Lechner, B. Watson, U. Wilensky, and M. Felsen. Procedural city modeling. In *1st Midwestern Graphics Conference*, St. Louis, MO, 2003.

[9] A. Lindenmayer. Mathematical models for cellular interaction in development – i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18:280–289, 1968.

[10] Mammoth Team. Mammoth massively-multiplayer online game. Website, July 2008. `http://mammoth.cs.mcgill.ca/`, last visited Jul. 18, 2008.

[11] B. B. Mandelbrot. Stochastic models for the Earth's relief, the shape and the fractal dimension of the coastlines, and the number-area rule for islands. *Proceedings of the National Academy of Sciences of the United States of America*, 72(10):3825–3828, 1975.

[12] P. Müller. *Design und Implementation einer Preprocessing Pipeline zur Visualisierung prozedural erzeugter Stadtmodelle*. Master's thesis, ETH Zürich, 2001.

[13] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23(3):41–50, 1989.

[14] J. Noel. Dynamic building plan generation. Technical report, University of Sheffield, 2003.

[15] J. Olsen. Realtime procedural terrain generation. Technical report, Department of Mathematics And Computer Science (IMADA) – University of Southern Denmark, October 2004.

[16] J. Orwant. Eggg: Automated programming for game generation. *IBM System Journal / MIT Media Laboratory*, 39(3 & 4), 2000.

[17] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[18] B. Pell. METAGAME in symmetric chess-like games. Technical report, University of Cambridge, Computer Laboratory, 2003.

[19] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[20] M. Powell et al. jMonkey Engine. Online, April 2008. `http://jmonkeyengine.com/`, last visited Jan. 28, 2009.

[21] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[22] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. Development models of herbaceous plants for computer imagery purposes. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 141–150, New York, NY, USA, 1988. ACM.

[23] J. Romein, H. Bal, and D. Grune. Multigame - a very high level language for describing board games. In *Proceedings of the first annual conference of the Advanced School for Computing and Imaging*, pages 278–287, May 1995.

[24] S. Smith. Adventure construction set. Electronic Arts, 1985. Video game.

[25] J. Tarbell. Substrate algorithm. Gallery of Computation website, April 2008. `http://complexification.net/gallery/machines/substrate/`, last visited Jul. 18, 2008.

[26] G. T. Toussaint. Solving geometric problems with the rotating calipers. In *Proceedings of IEEE MELECON'83*, pages A10.02/1–4, Athens, Greece, may 1983.

[27] Vivid Solutions. JTS Topology Suite. Online, December 2006. `http://www.vividsolutions.com/jts/jtshome.htm`, last visited Jan. 21, 2009.

[28] B. Watson. Modeling land use with urban simulation. In *ACM SIGGRAPH 2006 Courses*, pages 185–251, New York, NY, USA, 2006. ACM Press.

[29] W. D. Wells. Generating enhanced natural environments and terrain for interactive combat simulations (genetics). In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 184–191, New York, NY, USA, 2005. ACM.

[30] Wikipedia. Adventure Construction Set. Online, 2008. `http://en.wikipedia.org/wiki/Adventure_Construction_Set`, last visited Nov. 25, 2008.

[31] Wikipedia. Geneva. Online, 2009. `http://en.wikipedia.org/wiki/Geneva`, last visited Feb. 1, 2009.

[32] Wikipedia. Great Britain. Online, 2009. `http://en.wikipedia.org/wiki/Great_Britain`, last visited Feb. 1, 2009.

[33] Wikipedia. The Elder Scrolls IV: Oblivion. Online, 2009. `http://en.wikipedia.org/wiki/The_Elder_Scrolls_IV:_Oblivion`, last visited Feb. 1, 2009.

[34] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. In *ACM SIGGRAPH 2003 Papers*, pages 669–677, New York, NY, USA, 2003. ACM Press.