Persistent Realtime Building Interior Generation

By
Evan Hahn

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

September 2006

Your file Votre référence
ISBN: 978-0-494-18353-3
Our file Notre référence
ISBN: 978-0-494-18353-3

Abstract

We present a novel approach to generate virtual building interiors in realtime. The interiors are generated in a top-down fashion using architectural guidelines. Although a building interior in its entirety may be quite large, only the portions that are needed immediately are generated. This lazy generation scheme allows the use of only a fraction of the memory that a model of the entire interior would take. Our method provides realtime frame rates, making it attractive for interactive applications such as videogames and simulators.

Memory is controlled by deleting regions of the interior that are no longer needed. Any changes made in these regions is not lost as we provide a simple and efficient method to allow changes made to the interior to persist past the lifetime of the regions that contain them. This allows a dynamic, consistent environment and increases control over the content by allowing developers to make changes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The focus of this thesis is the development of a system that allows synthesizing non-existent building interiors during runtime. The generated building interiors are kept persistent, meaning that unneeded portions of the interior can be removed without eliminating the any changes made to these portions. Our system is intended for use in interactive 3D applications like video games, virtual worlds and simulators. Its main purpose is to help applications provide interactive environments that are much larger than the available memory and much larger than developers would be capable of creating manually in a reasonable amount of time.

Hand-held gaming systems and cell-phones would particularly benefit from this, since they have limited available memory. For instance, one of the most recent hand-held gaming systems, the Sony PSP, has only 32 MB of RAM[27]. This is an insufficient amount of space to hold many of the buildings that we tested in their entirety. The largest buildings we tested had an average size of 42 MB. Using our lazy generation approach however, even these larger buildings can be explored on a PSP with plenty of room to spare. The maximum memory used to explore our largest buildings averaged around 2 MB.

In addition to memory savings, generating environments procedurally can take substantial workloads from the hands of artists. This allows larger environments to be created in a shorter periods of time. In the case of realtime generators like our building interior generator, initial unedited environments can be provided almost

instantly.

There are also possibilities that are not available when manually creating content. By allowing the computer to create the environment, it is possible to provide a near infinite quantity of unique environments. This can greatly increase the longevity of an application. For instance, if used in entertainment applications like video games, the generator can provide new environments for the user each time it is played. A notable example of this can be found in the video game created by Blizzard Entertainment, Diablo II [7], which provides two dimensional randomly generated wilderness areas and dungeons. This award winning game was released in the summer of 2000 and remains popular to this day. In fact, this game has had such a large base of steady players that Blizzard has been releasing updates and patches for over five years after its release. While this game's longevity cannot be attributed to its procedural environments alone, they definitely decrease the monotony of constantly replaying the game.

Another possibility with procedural approaches is that environments of near infinite extent can be created. A demonstration of this was created by Greuter et al. [10]. They created an application that generates cities with a nearly infinite extent. Their approach will be described in more detail in section 2.2.

Procedural environments can provide substantial memory benefits as well. Generated environments are typically much larger than the data used to generate it. As long as the generated result is deterministic, this data can be stored instead of the entire environment. This can significantly decrease the required memory to permanently store the environment and the time it takes to transfer the environment over a network.

The required memory to use the environment can also be reduced. If the environment is generated in a lazy fashion, only the needed parts of the environment will be generated. Depending on the environment, the needed portions of the building may be much smaller than the total available environment. Lazy generation is particularly effective in interior environments like those found in the building interiors that our generator creates because at any given time most of the building is occluded by the walls and floors of the building. Our test results support this. For example,

the largest building we tested had 16127 regions. During an automated traversal through the entire building we found that the maximum number of visible regions any time was 42. This means that during the walk, roughly 99.7% of the building was occluded. Figure 1.1 shows three views of this building. The left most shows a first person view of the building. The middle view shows the portion of the building that was generated to display that view. Finally, the rightmost view shows the same building generated completely.

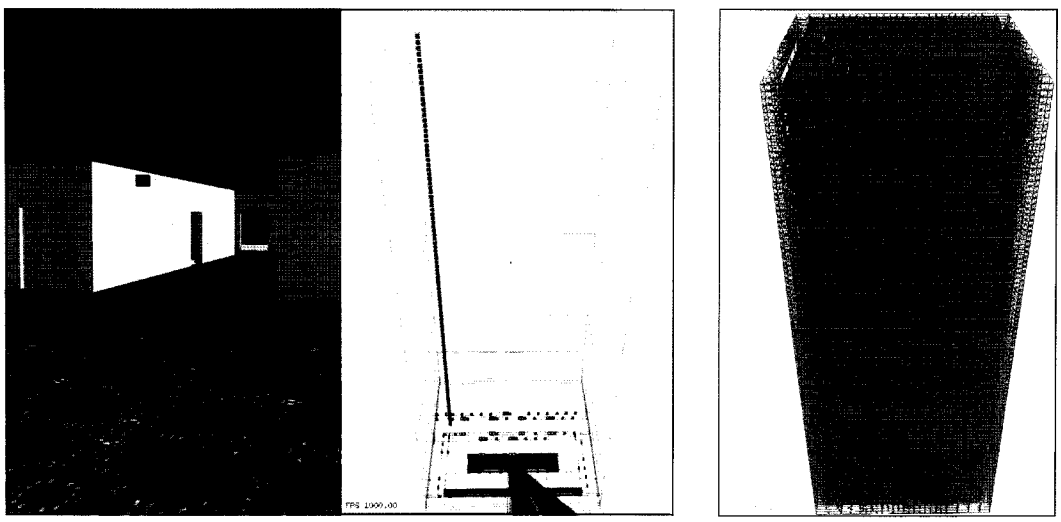


Figure 1.1:
Left: A screen shot showing a building's first person view and its generated contents. Right: A wireframe view of the same building generated completely. This building has 100 floors and 16127 regions.

In order to save memory from lazy generation, parts of the environment must be deleted when they are no longer needed. Unless care is taken, any changes made to the deleted parts, will also get deleted. We have developed a persistent change manager to solve this problem. Its main purpose is to provide an efficient way to keep track of changes after regions are deleted. This allows any changes made to persist past the lifetimes of the interior regions that contain them.

Without our manager, we would either have to disallow changes to the environment or accept the fact that changes may be lost unpredictably. We believe both of these options are unacceptable for interactive applications. The first option would

greatly limit the interactivity and realism in the environment. Buildings typically contain many moveable and changeable objects, like doors, furniture, books, and so forth. With so many possible changes to be made, it would feel unnatural if nothing could be changed at all. Interactivity would also suffer since the results of interactions would never be able to change the environment. The second option has the potential to be confusing and frustrating for users. For instance, changes to the environment often make excellent navigational landmarks for a user. Removing these landmarks without the users knowledge can be extremely confusing. It can also be frustrating when users spend time to make changes, and have these changes disappear unpredictably later on.

A secondary purpose for the persistent change manager is to allow pre-release developer changes. Control is one of the greatest things a developer sacrifices when procedurally generating an environment. This is especially the case when they are generated during runtime, since it is not normally possible to edit the results. Our manager, however, puts some control back into the hands of the developer by allowing a number of changes to be made. Our manager can efficiently apply these changes as needed during generation.

1.1 Motivation

Much of the inspiration for this generator came from urban sandbox games like the Grand Theft Auto Series [24]. Although there often are enterable buildings in these games, the vast majority of the buildings cannot be entered. This is most likely due to the shear number of buildings in a typical city. Given limited development times and memory requirements, it is infeasible for developers to create such a large number of building interiors. In many types of games it is acceptable to limit where a player can explore, but a major appeal of sandbox games is the freedom they provide. Players get a good sense of freedom already, but imagine how free a game would become if every visible building could be entered and explored. Imagine how real a game would feel if the player knew that something always existed on the other side of a door or window. Environments such as these become more feasible if large portions of it can

be generated during runtime. This is what our generator aims to do.

The novelty of this research is another motivating factor. We are not aware of any other lazy approaches to generate building interiors, and we have not encountered any research that provides persistent change management in interactive procedural environments.

1.2 Scope

Our hypothesis is that large skyscraper style building interiors can be generated on the fly, with realtime speeds. The structure of the generated buildings should be realistic enough to resemble real buildings in a first person view, and should also be suitable for interactive applications. To enhance interactivity and realism, we also explore persistent change management in a procedural environment.

1.3 Contributions

The contributions of this thesis are:

- A lazy generation technique to create interactive building interiors in realtime [11, 12]. Using this technique can significantly decrease an artist's workload in applications that require large building interiors. We are not aware of any other techniques to generate building interiors in a lazy fashion.
- Persistent change management in a procedural environment[11]. This allows the lifespan of a change to the environment to be independent of the lifespan of the environment itself. To our knowledge this the first research to add these capabilities to a procedural environment.
- An implementation of the above techniques.

1.4 Thesis Overview

This thesis is divided into seven chapters. Chapter 2 describes the work related to synthesizing geometry, generating architecture and persistent changes. In Chapter 3 we explain how the system generates building interiors in a lazy fashion. Next, Chapter 4 covers how we manage our memory by freeing up the space used by unneeded portions of a building. Chapter 5 then covers how we manage persistent changes, and the techniques we used to provide four different types of changes. Finally, in Chapters 6 and 7 we present our empirical results and conclusions.

Chapter 2

Related Work

This thesis is concerned with two major topics: Generating building interiors, and managing persistent changes. Related work for generating building interiors is reviewed in Section 2.1. This section starts by covering broad procedural geometry paradigms and common techniques that fit into these paradigms. It then proceeds to cover more specific approaches to generate geometry in realtime. Section 2.2 covers related approaches to generating architecture. Finally, work related to persistent change management can be found in Section 2.3.

2.1 Procedural Geometry Synthesis

We are concerned with synthesizing the geometry of a building interior in realtime, using minimal human input. Algorithms exist to generate a wide variety of objects and environments, including plants, terrain, forests, buildings and cities. In spite of this variety, algorithms for procedural geometry synthesis normally fall into two classes[14]: data amplification algorithms[21, 22, 23, 25, 26, 33] and lazy evaluation algorithms[2, 6, 9, 10, 13].

The term data amplification was first used by Alvy Ray Smith[26] to describe the transformation of small amounts of data into massive amounts of geometry. A popular example of a data amplifier is the L-system. L-systems were first created by Aristid Lindenmayer [21] to mathematically model the development of filamentous

organisms, and was later developed into a full system for modelling the behavior of plant growth by Prusinkiewicz and Lindenmayer [23]. L-systems are a very popular method for describing procedural models. They are grammars of string rewriting rules where each production is applied in parallel. The final result of these grammars are strings that are independent of the order that the rules are applied. These strings act as intermediate representations that can be interpreted to produce geometry for rendering. L-systems are most commonly used to model plants and other natural shapes[14], but some man-made objects can be effectively described using L-systems as well. For instance, Parish and Muller [22] use extended L-systems to model the road networks of cities.

Procedural methods that are classified as data amplifiers typically create intermediate representations like scene descriptions. After an intermediate representation is complete, it is then sent to the renderer to create an image. The intermediate representations can often become extremely large. In the case of L-systems, the intermediate representation can grow exponentially with the number production applications. These representations also often lack any useful organization, which makes it difficult to render the geometry efficiently.

Data amplifiers generate all of the geometry described by their representations. This is in contrast to lazy evaluation techniques, since lazy evaluators only generate the geometry needed to render an accurate image. This avoids the need to create and process massive intermediate representations to render images. Since only the geometry that is needed should be generated, the ability to cull away significant portions of a scene is of great importance to methods in this paradigm. Our building interior generator falls in this paradigm, and efficient culling is achieved through the use of portal-region graphs (see section A.3). Other data structures for efficient culling include grids, octrees, and bounding volume hierarchies. [13]

An example of a lazy evaluation method is procedural geometric instancing, developed by John C. Hart[13]. It was designed for efficient rendering and to make the articulation of the representation easier. It is based on augmenting instances in a scene graph with a procedure. A scene graph is a common representation in computer graphics used to organize the logical and spatial characteristics of a scene. Instancing

is used to save memory when placing models in scene graphs. Each instance placed in a scene graph is a transformed copy of a master model. With procedural geometric instancing, an instance can be augmented with a procedure that will be executed each time the object appears in the scene. This procedure, for instance, can generate the object if it does not already exist. Hart organized the objects of the augmented scene graph into bounding box hierarchies that can be used for efficient rendering and to determine whether something needs to be generated. The greatest difficulty with this approach is finding the bounding volumes of procedural geometry without actually generating the geometry. Our approach is not faced with this difficulty since our generation steps start out with bounding volumes and generate the contents contained in them.

Many environments that require long periods of time to generate completely are possible to generate in realtime through the use of lazy generation. This is because lazy generators often only need to generate a small portion of the entire environment to simulate and render the scene.

A realtime lazy generator for terrains was created by Ammeter and Mikhailyuk in order to reduce dependency on geographical databases requiring large amounts of storage space[2]. Their terrains consist of fixed sized square blocks arranged in a grid pattern. Lazy generation is achieved by generating the squares that intersect with the camera's view frustum. These squares are placed in a list that is updated every frame. Squares that are not intersecting the frustum are removed from memory to save memory and to avoid unnecessary rendering.

Each square block contains a regular grid of height values that are stored in an array. The height values are determined using a randomized recursive subdivision technique called midpoint displacement. Each step of this process uses two previously set points and displaces the height of their midpoint to a random value. This value will fall between the two heights of the previously set points. After the block is generated it may be reduced to a lower level of detail by joining triangles. Joining is determined using the horizontal distance from the camera and the vertical amount that the common vertex moves after the join.

Another generator that is similar to the above, but much larger in scope, is presented by Dollins[9] in his Ph.D. thesis. He describes a system to create interactive large-scale worlds in realtime. His system generates a multi-resolution description of the geometry and behaviors on the fly, based on the limits of the user's view. The terrain of the world is created by using quad-tree subdivision, where the terrain shape is defined by the midpoint of each cell. A cell's midpoint is derived using a function of its nine parent layer cells. Dollins uses a pseudo-random number generator to randomly offset the terrain's vertices and to generate other objects that may be in the cell. To keep the environment consistent, he seeds the random number generator for each cell by combining the x and y coordinates of each cell. Our system uses a similar approach to seed the random number generator, which is described in section 3.11.

Similar random seeding techniques are also described by Lecky-Thompson[18]. This book contains various techniques that can be used to help generate 'infinite' game worlds. Some of the infinite world generation techniques of this book have been applied in a generator by Greuter et al.[10]. They have developed an approach to generate 'pseudo infinite' cities in realtime. Their approach generates pseudorandom building exteriors as they become visible to the user, and temporarily stores them in a least recently used cache. Items of this cache are deleted either when the cache exceeds its limit or when buildings exceed a specified maximum age. We are also using a least recently used cache to help manage our memory. See section 4.2 for details.

To keep such an enormous environment consistent, they seed the random number generator for each building they generate using the position of the building. The buildings created by their generator are placed in a regular grid arrangement. To generate the city's building exteriors, the generator first creates a 2D base for the building by randomly overlapping randomly selected polygons. The generator then makes the buildings 3D by extruding the base polygons to random heights.

Unfortunately, only the exteriors of the their buildings are generated, so users are limited to exploring the streets. However, another generator called the Descensor Engine exists that generates cities in realtime with complete buildings. This generator was created by a company named Binary Worlds [6]. Their engine generates 3D

worlds containing a landscape, trees, roads, houses, and multistory buildings. Simple interiors are even generated within the structures. They have not disclosed much information about how they generate their worlds, so we are unaware of how close their approach to generate buildings interiors is to ours. Nevertheless, we do know that our generator achieves realtime frame rates while generating building interiors that are many times larger than the interiors found in the demo they released.

Throughout the course of our research we have not found any other approaches that generate building interiors in realtime. However we have found one approach that is meant to generate interior environments before they are used, during runtime. This generator was created by Roden and Parberry[25], and they demonstrated their approach by generating dungeon style game levels during runtime. This generator is based on placing prefabricated geometry into a uniform 3D grid. This generator requires much more human input than ours because the prefabricated geometry requires human modelling. Using this geometry, the generator creates the level in three stages.

The first stage creates a 3D graph based on the grid to decide the layout of the level. To create the graph, a grid cell is selected to place the entrance node in. This cell can have up to four adjacent grid cells, and the connections to these are put into a list to be used at a later time. Additional steps are randomly selected from this list to create and connect new nodes. Once a connection is selected, the generator checks to see if it breaks any constraints. If it does break a constraint, a new connection is selected at random from the list. The generator terminates if a valid connection cannot be found. If a valid connection is found, a new node is created and more connections are added to the list for further generation steps

The second generation stage places the prefabricated geometry in the grid. Selection and placement is based on the nodes and connections of the graph. After this is completed, the third stage adds content like artificial characters, items, and other details to the level.

Their generator also creates the data needed for efficient rendering and collision detection. For occlusion culling, they place portals between the cells of their level. Our generator also uses portals for this purpose. Our method is described in appendix A.3.

For efficient collision detection they construct a binary tree of axis aligned bounding boxes. Our collision detection method is described in appendix A.1.

2.2 Generating Architecture

This section covers related works in designing and generating architecture that are not used in realtime generators. "A Pattern Language"[1] by Christopher Alexander is a book containing a pattern language to aid the design and construction of towns and buildings. This book has a broad sequence of patterns covering the design of regions and towns, neighborhoods, clusters of buildings, buildings, rooms and alcoves, and the details of construction. Each pattern of the language describes a frequently occurring problem and a solution that can be adapted to a designer's needs. These patterns are most effective when combined with other patterns. In fact, almost all of the patterns are connected to other patterns in the language. This lets designers choose whole sequences of patterns that can be combined to produce designs.

This pattern language is meant to be used by humans to generate effective designs. Most of the problems and solutions are kept broad and general, allowing them to be adapted to the designer's particular needs. This however makes it difficult to directly use the patterns in a computer generator. In spite of this, the language still provided valuable help in producing realistic building interiors. It provided insight into the design of building interiors, and some of the rules we defined to implement our generator were derived from the patterns of this book.

Formal grammar approaches to aid the design of buildings have been developed that can be used in computer generators more readily. George Stiny[28] developed shape grammars to allow the definition of various architectural styles. Unlike traditional grammars which process symbols, shape grammars process shapes directly. Two steps are needed to use a shape grammar: shape recognition, and rule application. A rule of a shape grammar describes a start shape and a new shape to replace it with. After rules are applied new shapes often emerge that rules can be applied to.

The emergence of new shapes can make computer implementation and automatic derivation difficult. Stiny[29] has also created a more computer friendly subset of

shape grammars called set grammars. Set grammars treat shapes as symbolic objects, so they do not require shape matching procedures.

Using the ideas from shape and set grammars, Wonka et al.[33] describe a framework to allow the automatic generation of architecture. They have demonstrated its use in generating building exteriors. Their framework consists of a database containing two types of grammars: a split grammar and a control grammar. Split grammars are a specialized type of set grammar with two types of rules: split rules which split shapes into smaller shapes, and conversion rules which transform single shapes into other shapes. Control grammars handle the spatial distribution of the shapes in an orderly fashion.

Although we have not defined any explicit grammars in our system, most of the building is generated by splitting temporary regions into smaller regions of various types, or by replacing temporary regions with other types of regions. These actions resemble the two types of rules in their split grammars, so it should be possible to define a split grammar that captures the transitions between our region types. We summarize the transitions between the region types in a graphical fashion in section 3.12 that resembles a split grammar. These transitions however, make up only a small portion of our generator. The generator also needs to know how to select the transitions and all of the details in constructing the regions involved with the transitions. This is the main reason we are not using external grammars with our system. Doing so would greatly increase the difficulty in implementing our generator, and would have little benefit unless the selection and construction details were also made external. Also, by keeping everything internal to our code we are able to specialize the generator for a more simple and efficient implementation.

2.3 Persistence

Persistence concerns the length of time for which data exists and is usable[3]. This is a common topic of research in programming languages. Persistent programming languages such as PS-Algol[3] aim to make the use of data independent of its persistence. This independence, for example, allows data in a database and data in a

temporary variable to be handled with the same code. Persistent object stores were created to help implement these languages[8]. They attempt to abstract persistence from the physical properties of data such as storage size, speed, and stability.

Another area where persistence is a concern is that of collaborative virtual environments. Collaborative virtual environments are virtual worlds shared by participants across a computer network. Participants in these worlds are able to interact with its contents and communicate with each other using a variety of media [4]. In these environments, persistence deals with the extent that a virtual environment exists after its participants have left[19].

An example of a collaborative virtual reality system is the V-Worlds project created by Vellon et al. [30]. Their system provides a long term, persistent, changeable world. That is achieved by automatically logging changes to object properties. When a property is changed, the server automatically records it in a sequential log file. Not all changes will be logged, however. To avoid unnecessary logging, V-Worlds allows properties that should not be logged to be marked as volatile. The log files can become quite large if enough changes are made. To avoid this, V-Worlds can also write out its entire state to a new log file. The old log file can then be archived or deleted.

To restore the state of the world, V-Worlds sequentially re-applies all of the changes recorded in the log. This allows the world to be persistent even if the server crashes.

In procedural environments, persistence has a similar meaning, but since static environments can be consistently regenerated, persistence in procedural environments is concerned mainly with the length of time that the changes to the environment exist. There are many factors such as memory and processor power that can limit the lifetimes of changes, but we are only concerned with one: The lifetime of the environment itself.

The reason this is a major concern for realtime procedural environments is that unneeded sections of the environment are often removed from memory while users travel through the environment. This can cause changes to be removed prematurely and unpredictably. To compound the problem, removed sections will cease to exist anywhere because the environment is generated in realtime. This makes saving

changes more difficult because the changes cannot be stored with the sections of the environment that they lie in.

We have developed a way to make the lifetimes of these changes independent of the environment they lie in. This allows the lifetimes of changes to be controlled in a predicable manner. We do not know of any other realtime generated environments that provide this capability.

Chapter 3

Lazy Building Interior Generation

This chapter describes our approach to generating building interiors in realtime. The chapter begins by describing the composition of our interiors and how they are generated in a lazy fashion. The generation rules, temporary region types and portals types are then presented. To simplify the implementation, generation has been split into a number of stages, and these stages are described in detail in Sections 3.5 through 3.10. Section 3.11 then describes how the systems generates pseudorandom numbers in order to produce consistent buildings. Finally, Section 3.12 provides a summary of how all the region types relate to each other.

Our approach is designed to generate skyscraper style building interiors, focusing on generating the structure of the interiors. Thus, exterior elements such as window placement and building shape are ignored. Realistic object placement is also out of the focus of our generator.

This generator adopts a lazy approach to procedural generation, meaning that only the needed portions of the interior will be generated at any given time. This approach generates interiors in a recursive, top-down fashion based on splitting temporary regions into new regions. Adjacent regions of our interiors are connected using faces called portals. Descriptions of portals and the main region types of our interiors can be found below:

Portals are rectangular faces used to specify areas where two regions are connected. These are used to guide generation, for quick visibility calculations, and to allow

objects to travel between regions. Portals can be assigned a number of types to guide the generator. See Section 3.4 for descriptions of these types.

Temporary regions are regions of space where generation can occur. They serve as placeholders to indicate what parts of the building have not yet been generated. In our implementation these regions are represented using axis aligned bounding boxes. There are many different types of temporary regions, and each of them give different generation results. See Section 3.3 for details.

Built regions are the final visible product of the generator. These regions hold the geometry needed for rendering and collision detection, as well as any visible objects that may be placed in the building.

The portals and regions mentioned above are the main elements composing our building interiors. Details about how the interiors are generated in a lazy fashion can be found in the next section.

3.1 Lazy Generation

Initially, a building interior consists of a single temporary region, and one or more portals. The temporary region specifies the volume where the building will exist and the portals are used as entrances and windows.

Generating our building interiors is a recursive process. At each generation step a temporary region is either split into smaller temporary regions or replaced with a built region. This will continue until there are no more temporary regions where generation is needed. After each temporary region is split, portals may be created to connect the new regions. These portals must only connect to the newly created regions or consistency errors could result.

Points are used to limit generation to needed regions. Given a set of temporary regions and a point, generation is only carried out in the region that contains the point. The first region to contain the point may be split into smaller regions, and

the process only continues with the smaller region that contains the point. It stops when the point lies within a built region. Figure 3.1 shows an example generating a section of a building around a point.

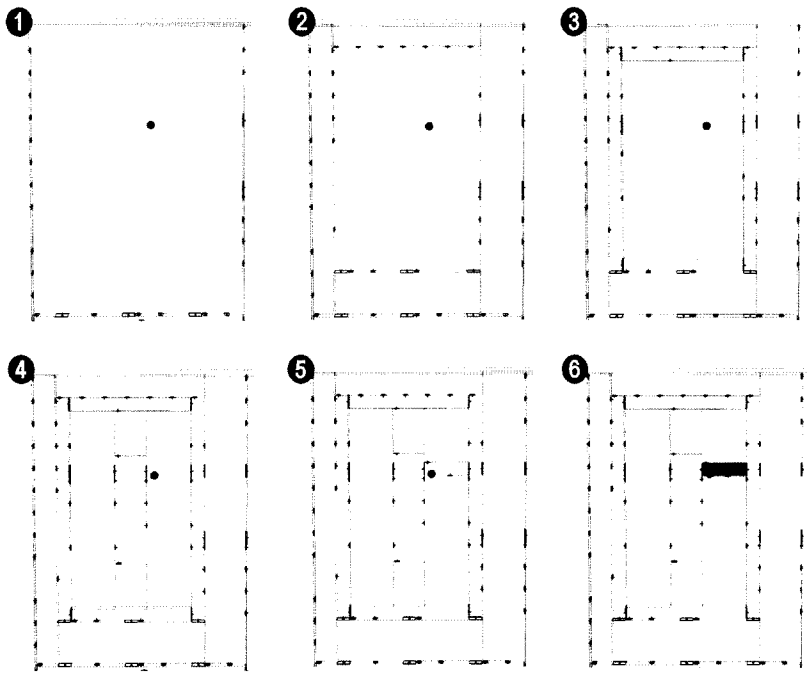


Figure 3.1: An example generating a section of a building around a point. Step 1 shows the initial region and a point that generation will be based on. The point is in a temporary region, so generation occurs, creating the regions in step 2. Next, the generator will find the new region that contains the point. Since this new region is also a temporary region, the generation will occur inside of it. Similar steps will be followed until step six is reached. The region that contains the point in this step is a built region. This will halt the generation process.

Temporary regions are generated around points whenever access to a non-existing built region is required. For instance, the camera is initially placed in the building by generating in the initial temporary region around the camera position. Generating around a point is also done when an object moves across a portal into a temporary region.

This technique is even used to generate the other visible regions in the building. After the system renders a region it checks to see if any portals are visible. If a visible portal has a temporary region on its opposite side, this temporary region will generate around a point from the portal to create a built region. The system will then render the new built region and the process will continue until only built regions are visible.

Building interiors can normally exploit occlusion to a great degree which makes lazy generation particularly effective. However, this also makes the order that regions are generated depend on the path taken through them. To prevent this from causing consistency errors, we designed our generator so that the results are independent of the order that existing regions are generated.

When a temporary region generates, it must do so independently of any other existing region. This means it cannot use any information from other existing regions, and it cannot alter any existing regions. Since the generation result is independent of the rest of the building, it doesn't matter what order the existing regions are generated to produce a consistent building.

3.2 Generation Rules

We developed a number of rules to follow when implementing this generator. The first four rules are mandatory and must be followed or errors may result. Possible errors may include visual artifacts, and buildings that are not consistent each time they are visited. These rules are written in italics below. The remainder of the rules were created in order to produce more realistic buildings. The generator will still work properly if these rules are not followed, so it may be acceptable to implement algorithms that aren't guaranteed to follow these rules all the time but are much faster than algorithms with such a guarantee.

1: ***Generated contents must never protrude out of the bounding volume of the temporary region that generates it.*** If this were to happen the protruding portions would likely intersect with other parts of the building.

This would cause visual artifacts if both regions were visible at the same time.

2: ***Existing regions of the building should never overlap each other.*** It is unrealistic for two regions to occupy the same space at the same time, and like the first rule, visual artifacts could also occur if this rule is not followed. Note: we have found it useful to make one exception to this rule when placing elevator shafts. See Section 3.5.2 for details.

3: ***When generating, a temporary region must not use information from any other existing region.*** If this rule is not followed the building would not be guaranteed to be consistent each time it is explored. For instance, when generation occurs in a temporary region, its neighboring regions are not guaranteed to be in the same state each time. If the generated results were based on information in a neighboring region they would also not be guaranteed to be identical.

4: ***When generation occurs in a temporary region, no other existing regions can be altered.*** This includes the placement of portals and objects as well. No other existing regions can be changed at all. Consistency errors could result if they are, because the order that regions are generated, if at all, is not guaranteed to be the same for each walk-through.

5: **Every region should be accessible.** Inaccessible regions are unrealistic and create areas of dead space that will confuse users that are trying to explore the entire building. Figure 3.2 gives an example of dead space. Note that our implementation currently has a bug where inaccessible regions are created. This however, is a rare occurrence and it is difficult to notice from a first person perspective.

6: **All significant portions of space should be used.** Space is rarely wasted in real buildings, and large portions of wasted space can cause the same confusion that inaccessible regions cause.

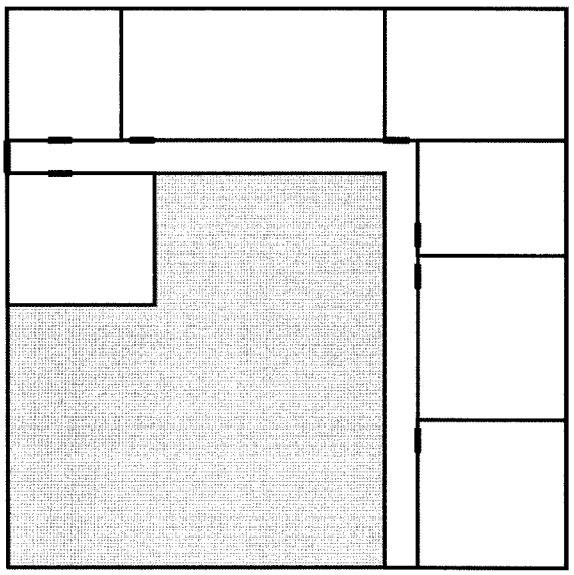


Figure 3.2: An example floor with a large portion of dead space. The thicker black lines indicate portals, and the grayed out area indicates where the dead space is. Dead space can be caused by creating inaccessible regions, and by not utilizing all significant portions of space.

7: **In most cases private rooms should be accessible through a public room or a hallway.** This was inspired by the "Intimacy Gradient" design pattern found in [1]. Currently in our buildings, every room is considered to be private, so to enforce this rule, every room must have hall access. We limit the minimum dimension of room clusters to ensure this. The minimum dimension of clusters at the building perimeter should be one average room thick, and non-perimeter clusters should have a minimum dimension of two average rooms thick. The size of an average room is defined by a constant in our implementation. Figure 3.3 shows an example floor with four properly sized room clusters.

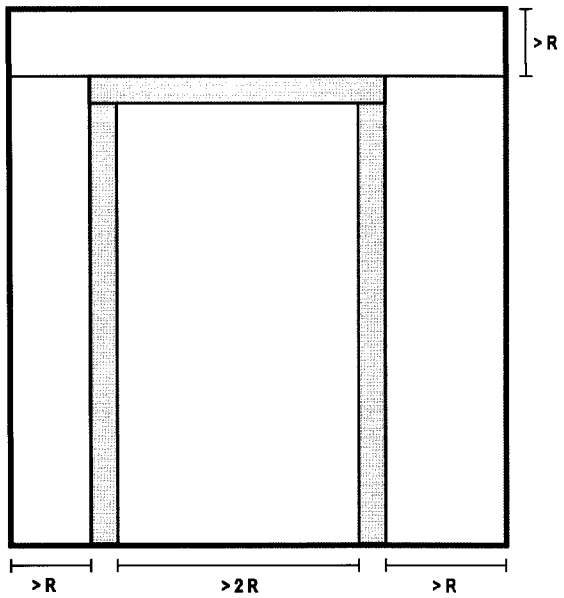


Figure 3.3: An example floor demonstrating the proper widths for room clusters. The gray regions represent the hallways, and the remaining regions represent room clusters. R is a constant specifying how wide an average room should be.

8: **Parallel hallways should have rooms between them.** Parallel hallways are redundant if they are right next to each other.

9: **If possible, hallways that touch other hallways should be connected.** This improves the circulation of the hallways, and reduces the number of hallway dead ends.

10: **In large enough regions, hallways that follow a loop shape should be favored.** We found through experimentation that this rule helps creates hallways that appear more regular and realistic. In our implementation, a large enough region is defined to be at least 5 average rooms thick.

3.3 Temporary Region Types

Rather than developing one type of temporary region that can generate every part of a building, we created seven different specialized types for different tasks. These types are outlined below:

Building: The initial temporary region of the building is given this type. These temporary regions create elevator shafts, stairwells, and other global aspects of the building like the textures to be used in the building.

Floor Divide: Temporary regions of this type are used to divide the building into uniformly spaced floors. Each division occurs at the floor closest to the middle of the region, and this creates two new temporary regions. If any of these new regions cannot be further divided into floors they are set as the *Hall Divide* type. If not, they are set as the *Floor Divide* type.

Hall Divide: This type of temporary region creates either a *Hall Loop* temporary region or a *Hall Segment* temporary region depending on its dimensions. (See rule 10 of section 3.2.) This new region is created inside the boundaries of the old region, and the remainder of the space in the old region is divided to create at most four more bounding box regions. These regions will either be *Hall Divide* regions or *Room Cluster Divide* depending on their dimensions.

Room Cluster Divide: These temporary regions divide to eventually create individual rooms. Each division divides the temporary region into two new *Room Cluster Divide* temporary regions. Divisions always occur between two portals. If there is less than two portals, or it is not possible to create two reasonably sized regions, division does not occur. Instead, a built region room is created with its dimensions.

Hall Loop: This type of temporary region creates a rectangular loop of hallway. The loop consists of four *Hall Segment* temporary regions. The middle region will either be a *Hall Divide* type or a *Room Cluster Divide* type depending on its dimensions. (See rule 7 of section 3.2 for details.)

Hall Segment: This type of region creates a rectangular built region section of hallway. The dimensions of the built region will be the same as its parent

Elevator Shaft: In a similar fashion to the hall segment, this type of temporary region creates an elevator shaft. The shaft will be a built region with the same dimensions as its parent

3.4 Portal Types

To guide our generator we assign different types to the portals in the building. This allows the generator to quickly decide what to do with each portal it encounters, and generate appropriate geometry for each portal.

Entrance: This type is given to all door portals that connect the building to the outside world. Our current implementation only has one type of entrance, so all entrances are treated equally. In the future it may be useful to have different types of entrances, so the generator could better create geometry to suit the needs of the entrance. This would, for instance, allow the generator to create an extravagant lobby around a main entrance, and a clear hallway for an emergency exit. Since this type of portal is attached to the building exterior, it is not be placed by our generator. These portals must be placed either by a human or another generator before our generator starts.

Elevator: This type is given to elevator door portals. Since elevators usually connect to hallways, our generator makes sure that portals of this type always connect

to hallways.

Hallway: This type of portal is used to connect hallways together. In our implementation, hallway regions are always rectangular shaped. In order to get more elaborate hallways in the building, these rectangular regions are attached using *Hallway* portals. When a *Hallway* portal is encountered during generation, a hallway region should be constructed and attached to it.

Room Door: This type of portal is found where ever a door into a room is located. When a *Room Door* portal is encountered during generation, a room should be constructed and attached to it.

Window: This type of portal is used to indicate where windows should be placed in the building. Since windows are part of the exterior of the building, our generator will not place this type of portal. Therefore, these portals will have to be placed before generation begins in order for generated buildings to have windows.

3.5 Building Setup

Building setup occurs with *Building* type temporary regions. Every thing that affects the building globally occurs here. The two main tasks carried out here in our current implementation are texture selection and elevator shaft creation.

3.5.1 Texture Selection

Texture selection in our implementation is currently basic. Eight textures are chosen for the building interior. We select three textures for the floors, walls, and ceilings of hallways, and three separate textures for the floors, walls, and ceilings of rooms. Two textures are also selected for the doors, and elevator shafts. For each of these locations, we created a list of possible textures, and selection is done pseudorandomly from these lists. Currently, each item has an equal chance of being selected and no attempt is made to match the textures for aesthetic appeal.

3.5.2 Elevator Shaft Creation

Our generator gives elevator shaft regions special consideration. Normally regions existing at the same time never overlap each other. However, when elevator shafts are created, they are simply placed inside of the *Building* temporary region without splitting this region. This is done in deliberate violation of rule 2 of section 3.2. Although this rule is considered to be a mandatory rule, we have good reasons for making an exception in this case. These reasons are discussed below.

Since our temporary regions are axis aligned bounding boxes, the only way we can create an elevator shaft in the middle of the building without creating overlapping regions is to split the *Building* region up into smaller regions. This however would have serious effects on the resulting building. It would greatly reduce the variety of floors produced for a building, since the split would exist on every floor of the building. Also, since portals can only be placed between newly split regions, all of the portals connecting each region on every floor would have to be placed in this generation step. This would require portals to be placed prematurely, and an enormous number of portals to be placed at one time. Large numbers of portals should be avoided because they take long to place and also create more work for future divisions.

The elevator shafts are pseudorandomly placed in the interior of the *Building* region so that at least an average sized room can be placed between the elevator shaft and all of the edges of the building. After the elevator region is placed, *Elevator* portals are placed along the shaft in the proper positions for each floor. Our current implementation only places one elevator shaft that spans the height of the building, although it is possible to place multiple elevator shafts.

Since placing elevator shafts is an exception to generation rule 2, our generator needs extra help to avoid errors. Temporary regions keep track of elevator shafts so they can generate regions based on the shaft if needed. When a temporary region splits, a reference to the elevator shaft will be given to any new regions that intersect the elevator shaft. This allows regions to know if the elevator shaft intersects it in constant time.

While our implementation only creates elevator shafts, stairways can almost be handled in the same way. Stairways however, require a much greater amount of

geometry to be created, and are different from elevator shafts in terms of visibility. For these reasons, we expect it to be beneficial to generate elevator shafts in a lazy fashion similar to the way floor division is done. This could greatly cut down on the amount of geometry created at one time. It would also be worthwhile to instance a master copy of a group of stairs, since most of the geometry in a typical stairwell is repeated and can be easily reused.

3.6 Floor Division

Floor division occurs in *Floor Divide* temporary regions. Instead of dividing the entire building into floors all at once, only one division will occur, which will create two new *Floor Divide* regions. Creating the floors in this fashion is ideal for our lazy generation approach, since it eliminates a lot of unnecessary divisions. It also helps create a well structured generation tree, which we use for memory management (See Section 4.1) and point location (See Section A.2).

Each division will occur at the floor closest to the middle of the region. Assuming that the building height is evenly divisible by the preset floor height, this will produce evenly distributed floors if the entire building is generated.

Figure 3.4 shows the floor division locations necessary to create a floor in one of our lazy generated buildings. Only seven divisions were necessary to create this floor. If this building was completely generated, it would have one hundred floors.

Our implementation currently does not place portals between floors after the divisions, but doing so has the potential of creating interesting areas in the building. For example, this would make it possible to create areas with higher than normal ceilings, holes in the floor overlooking public areas of the building, and so on.

3.7 Hallway Division

Hallway division occurs in *Hall Divide* temporary regions. This type of division is responsible for creating all of the hallways in a building and the room clusters between them. It was also the most challenging type of division for us to design. The

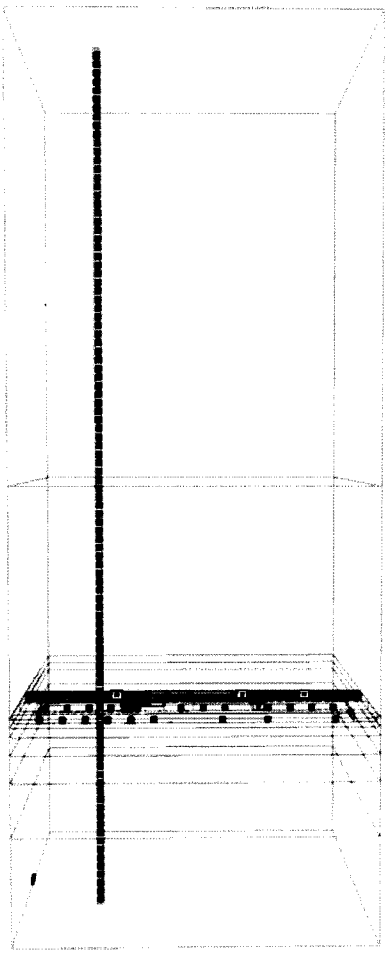


Figure 3.4: A side view of a lazy generated building showing the floor divisions necessary to create a single floor.

hallways produced by our early generators looked like random mazes, and took a lot of experimentation to create a generator that produces buildings that look like they could be real.

Hallway division is based on the position of a smaller temporary region that is placed in the dividing region. We will call this the hall region since it will either be a *Hall Segment* or a *Hall Loop*. The larger region is divided up around the hall region to create up to four more additional regions. This will happen if the hall region is placed in the middle, and one less region will be created for each edge of the hall

region that touches the boundaries of the region being divided. The surrounding regions will either be *Hall Divide* or *Room Cluster Divide* temporary regions.

The placement and dimensions of the hall region depend largely on its type. These details are be explained in subsections 3.7.2 and 3.7.3.

We divide the the *Hallway Divide* region by placing cutting planes on the edges of the hall region. Since the dividing region and the hall region are axis aligned bounding boxes, the new regions produced by these planes will also be axis aligned bounding boxes. This is necessary since our temporary regions are currently restricted to this shape.

The shapes of the outer regions depend on the order that the planes are used to cut the region. To simplify our implementation we use the same cutting order every time. The first two cuts occur along the shortest edges of the hall region, and the last two occur along the longest edges. This will produce regions with a similar shape to those in Figure 3.5.

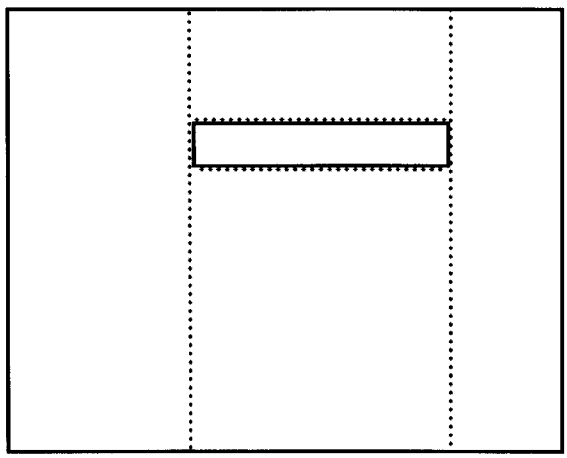


Figure 3.5: An example division around the hall region. The outer and inner solid boxes represent the parent temporary region and hall region respectively. The dotted lines represent the cutting planes.

We feel that this cutting order is the most versatile of the possible orders because it is symmetric in both directions and it is less likely to produce unusable skinny regions than some other cutting orders. We are achieving good results using this single cutting order, but more variety may be achievable if multiple orders were allowed. There are

even some ways to divide the region that are not possible using cutting planes. In the future it may be worthwhile to explore these possibilities to produce greater variety in our buildings.

Care must be taken when dividing up the temporary region so that none of the cutting planes intersect existing portals. Errors will result if this happens because portals need to completely lie on both edges of the regions it connects. To avoid this, our generator first generates the hall region without considering this problem. It then checks to see if any of its cutting planes are intersecting any portals. If there are any, it will move the edges with the offending cutting planes to the closest location that does not intersect any portals.

3.7.1 Portal Placement

The placement of portals has a huge effect on the outcome of the building. *Hallway* portals will cause the generator to create connecting hallways, and the placement of *Room Door* portals has major effects on the results of room cluster division. (See Section 3.8 for details.)

The generator frequently builds hallways connected to existing *Hallway* portals. When this is done however, we want to avoid breaking generation rule 8, which states that parallel hallways should have rooms between them. To help follow this rule we established a convention for placing *Hallway* portals along *Hall Segment* regions: Whenever we place a *Hallway* portal on the end of a *Hall Segment*, we face the portal away from the region, and whenever a portal is placed along the sides of a *Hall Segment* we face the portal into the region. Figure 3.6 shows an example of this convention in use. The dotted rectangle in the figure represents a *Hall Segment* and the thick black lines represent the portals. The arrows show the direction that the portals are facing.

With *Hall Loops*, we use a similar convention. The only difference is that *Hallway* portals will always face into the region no matter where they are placed.

This convention allows the generator to easily decide how to orient a *Hall Segment* when connecting it to a *Hallway* portal. If a *Hallway* portal is encountered that is

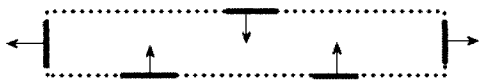


Figure 3.6: An example *Hall Segment* portal placement using our hallway portal placing convention.

facing away from the dividing region, the generator will know that the portal either is or will be attached to the side of a hallway. Therefore, in order to avoid breaking generation rule 8 the generator must create a hall segment that is parallel to the normal of the *Hallway* portal.

When placing portals, we must make sure that the generator does not place them on top of other existing portals. To simplify this, we maintain a list containing the intervals that portals can be placed in for each edge. This is implemented using a doubly linked list where each node contains either the beginning or ending of an interval. These will come from the extreme points of the edges and the portals that are placed on them. Since portals will never intersect, we do not have to keep track of whether a point starts or finishes an interval. Starting at the beginning of the list, each pair of points will give the beginning and end of an interval that a portal can be potentially placed in. Refer to Figure 3.7 for a demonstration of safe portal placement intervals.

Since portals will always be placed between two newly created regions, there will be no portals to consider when creating the lists. Thus, to initially create the interval list for an edge, all we need to do is place the two extreme points of the edge in the list. This pair of points will make up the start and end of the first interval in the list.

A list is updated each time a portal is placed on its edge. This is done by inserting the portal's extreme points between the points of the interval it is placed in. Since the list is implemented using a doubly linked list, and the location to insert will already be known, updating the list will take constant time for each portal placed.

Keeping a maintained interval list allows us to prioritize portal placement. We currently place portals along edges of a hall region in three phases of decreasing

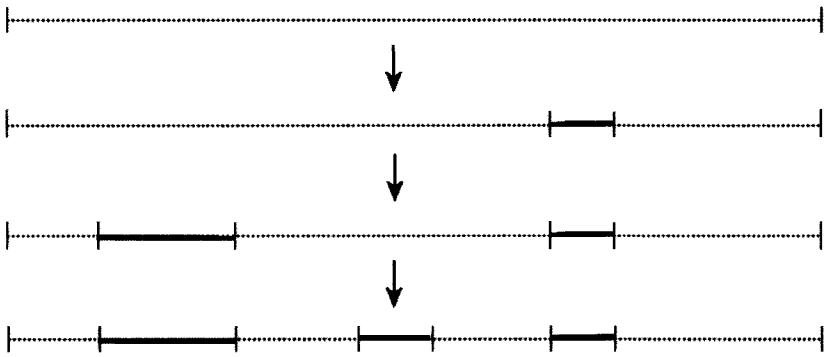


Figure 3.7: An example series of portal placements. The dotted line represents an edge, the thick horizontal lines represent portals, and the shaded areas represent safe intervals to place portals. The small vertical lines represent the points that will be placed in our interval list.

priority. The first phase places *hallway* portals directly across from other existing portals. The second places other *hallway* portals in between these, and the third places *Room Door* portals in the remaining space.

For the first phase, the generator checks to see if there are any *Hallway* portals directly across from the hall region. If there are, *Hallway* portals will be placed directly across from them on the hall region. Figure 3.8 gives an example of this. Placing portals in these locations helps to keep the hallways in the building well connected, and it also helps enforce generation rule 9, which states that hallways that touch other hallways should be connected. Before we introduced this step, we generated hallway floor plans with a lot of dead ends and a lot of unconnected touching hallways.

The second phase of portal placement will occur for each edge that touches a new *Hallway Divide* region. This phase will place more *Hallway* portals. Since all *Hallway Divide* regions create at least one hallway, placing these portals ensures that the future hallways can be connected to the rest of the floor's hallways.

Each safe interval of an edge's list will be visited during this phase. Ideally, we want room clusters that are at least two rooms thick between the hallways. If a interval is large enough to allow this, the generator will pseudorandomly place portals

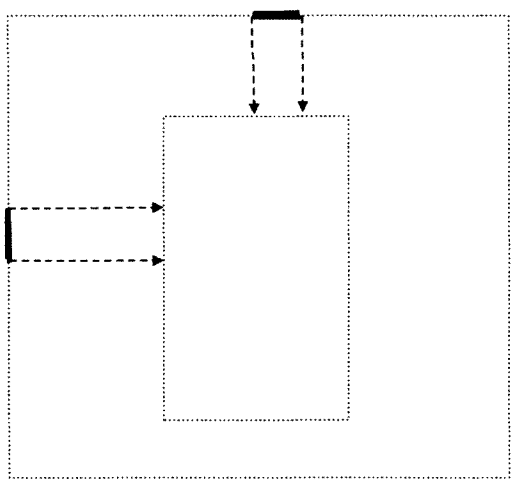


Figure 3.8: An example demonstrating portal placement when *Hallway* portals are found directly across from the hall region. The outer and inner dotted regions represent the parent temporary region and the hall region respectively. The thick black line segments represent pre-existing *Hallway* portals, and the dashed arrows show where new portals should be placed.

in the interval with at least two rooms worth of space between each. Pseudo-code showing how these portals are placed can be found below:

```
Given:
    PORTAL_WIDTH = the width of the portal to be placed
    TYPE = the portal type to place
    PLACE_PROB = the probability to place a portal
    PORTAL_DIST = the distance between portal placements
    PORTAL_COUNT = The count of portals already placed
    INTERVAL_LIST = the interval list
    INTERVAL_DIST = the distance from the start and finish of each
                    interval that portals should be placed
    GUARANTEED = True if at least one portal should be guaranteed
                 to be placed


for (all of the intervals in INTERVAL_LIST){
  PLACE_MIN = The first point of the current interval + INTERVAL_DIST
  PLACE_MAX = The second point of the current interval
              - PORTAL_WIDTH - INTERVAL_DIST

  while(PLACE_MIN < PLACE_MAX){
    if(a random number between 0 and 1 is less than PLACE_PROB){
      place a portal starting at PLACE_MIN with a width of PORTAL_WIDTH

      insert the portal's extreme points between the points of the current
      interval

      PORTAL_COUNT = PORTAL_COUNT + 1
    }

    PLACE_MIN = PLACE_MIN + PORTAL_DIST
  }
```

```
}

if (GUARANTEED is true and PORTAL_COUNT equals zero){
  For (all of the intervals in INTERVAL_LIST){
    if (there is room in the current interval){
        insert the portal's extreme points between the points of the current
        interval

        PORTAL_COUNT = PORTAL_COUNT + 1

        Exit loop
    }
  }
}
```

We adopted this approach to placing portals because it is simple and efficient, it provides the correct spacing between portals, and it gives a uniform look to the portal arrangement. There are many other possible approaches to portal placement, so in the future it may be worthwhile to explore alternate approaches to portal placement. If multiple approaches that produce good results can be found, it would be possible to provide more variety in our interiors by randomly selecting from these approaches.

The third portal placement phase will occur for all of the edges that touch a new region, regardless of whether it is a *Hallway Divide* or a *Room Cluster Divide* region. This phase will place *Room Door* portals along the edge so that future rooms will have access to a hallway. Placing these portals is done in a similar fashion to the previous phase except these portals are placed with at least one room worth of space between them.

After the generator has finished placing portals between the hall region and its surrounding, newly created regions, portals may be placed between the remaining newly created regions. Figure 3.9 shows the possible remaining edges to place portals along.

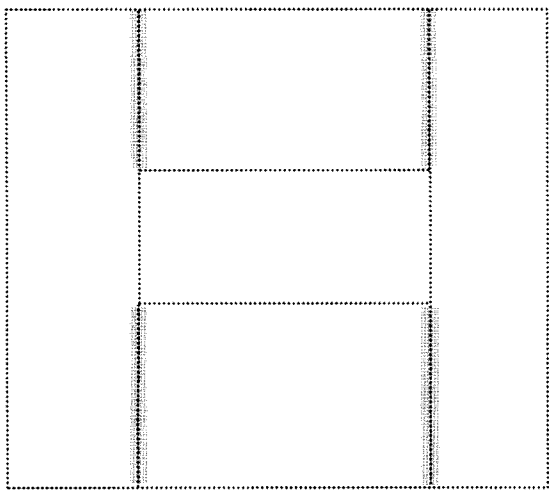


Figure 3.9: An example showing the possible remaining edges to place portals along. The dotted lines represent the boundaries of the newly created regions, including the hall region in the center. The shaded areas indicate the remaining edges to place portals.

Both *Hallway* portals and *Room Door* portals will be pseudorandomly placed on these edges. These portals will be placed in the same fashion as phase two and three described earlier to place portals on the hall region.

These portals are placed in anticipation that we will want the new regions to be connected later. Earlier versions of our generator did not place portals on these edges. The hallways created by these resembled a tree-like structure; the hallways branched out, and rarely reconnected to each other. Because of our lazy generation approach, the generator cannot wait until more of the building is created before it places these portals. (See generation rule 4 of section 3.2 for details.) Since, the generator did not connect these regions in the step that they were created, they became cut off from each other forever.

3.7.2 Hall Segments

Hall Segment regions are used to generate straight sections of hallway. A *Hallway Divide* temporary region will construct this type of hall region if its minimum dimension

is smaller than five rooms thick.

Hall Segment regions are constructed based on the pre-existing portals that are connected to its parent temporary region. If the parent possesses any *Hallway* portals, one of these portals will be chosen to build the *Hall Segment* off of. The choice of this portal is done pseudorandomly.

Building the segment is different depending on the orientation of the portal that is chosen. We first describe how to construct the hall segment if the portal faces out of the parent region:

Since the generator follows the portal placing convention we describe in section 3.7.1, if a portal is facing out of the parent region, the generator will know that the hallway on the other side of the portal will run along this edge of the parent region. Therefore, to avoid breaking rule 8, hallway segments built from this portal should run perpendicular to this edge. The width of the hall segment is specified as a constant in our implementation, and the length depends on the other portals in the parent region.

If the chosen portal has another *Hallway* portal directly across from it, the length of the hall segment will span across the parent region to connect the two portals. The probability of portals to be arranged like this is fairly high because portals are often placed in this arrangement by the generator. (See section 3.7.1 for details.) Connecting the two portals in this fashion promotes well connected hallway networks and discourages unconnected, touching hallways from forming.

If no *Hallway* portal is found directly across from the chosen portal, the length of the hall segment is calculated pseudorandomly so that it will be one room's thickness away from the edges of the parent region.

The hall segment is constructed differently if the chosen portal faces into the parent region. According to our portal placing convention this means that a hallway will run perpendicular to this edge. In this case, the generator will construct the hall segment to run along the parents edge.

We experimented with a number of ways to calculate the length of the hall segment, and generated the most consistently good looking results when the segment

spanned across all of the portals that share the edge of the parent temporary region that the segment is running along. Thus, our generator calculates the length pseudorandomly so that it will at least span across all of these portals.

Once the *Hall Segment* region is created we do a final check to see if there is sufficient space to create rooms along all of its edges. If an edge of the *Hall Segment* is too close to the edge of its parent region, the generator will expand the region till the offending edge is touching the corresponding edge of the parent region.

This is a rare occurrence with our buildings, but if the generator does not handle it, very small and awkward rooms could be produced. It could even be possible to generate rooms that the user cannot fit into. We feel that generating slightly wider halls is much more desirable over generating rooms this small.

3.7.3 Hall Loops

We occasionally produced good buildings using only hall segments to create the hallways. However, we also produced quite a few random and irregular looking buildings. We introduced hall loops in our buildings to solve this problem, because looping hallways are very common in real life building interiors.

Hall Loop regions are created in *Hall Divide* regions with a minimum dimension greater than 5 average rooms thick. As the name implies, *Hall Loop* regions create rectangular loops of hallways. The actual loop is not created when this type of region is created. *Hall Loop* regions give the bounding box of where the loop will get generated. Later, when generation is triggered in a *Hall Loop* region, the loop will get created. Figure 3.10 shows an example of the steps taken to create a hall loop.

The creation of the loop depends on its dimensions. In most cases, a middle region will be placed within a loop of hallways, but if the minimum dimension of this space is smaller than the minimum room width, a middle region will not get created. Instead, a *Hall Segment* region will be created with the same dimensions of the *Hall Loop Region.* This creates large open spaces that can be used as public gathering areas.

If there is enough space to create the middle region, five regions will be created: the middle region, and four *Hall Segment* regions to form the surrounding hall loop.

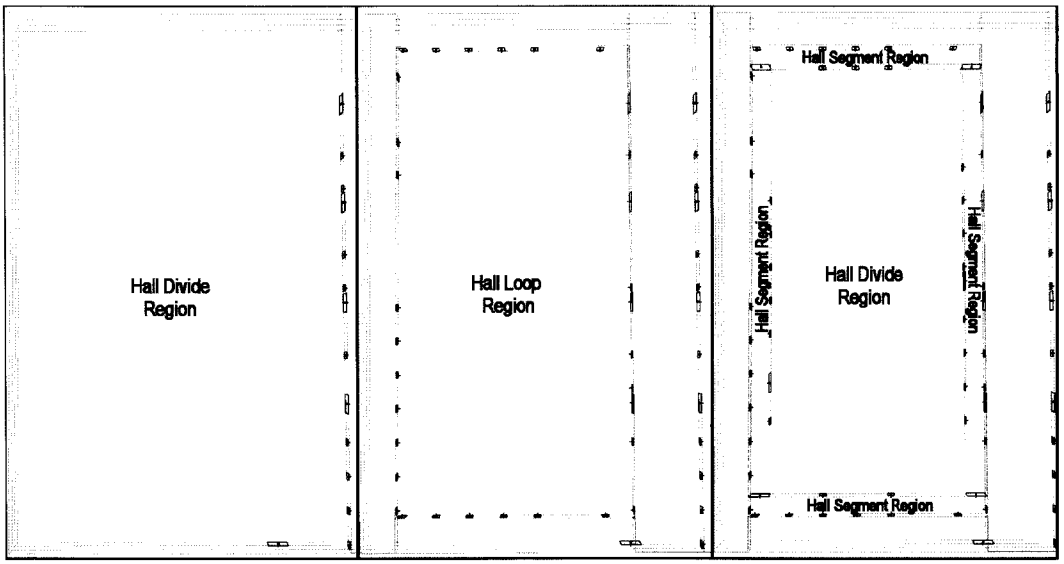


Figure 3.10: Example of the steps taken to create a hall loop. The *Hall Divide* region creates a *Hall Loop* region. Later, if generation is triggered in the *Hall Loop* region, The hall loop will be formed out of four hall segments.

The type of middle region also depends on its dimensions. If its minimum dimension is less than 3 rooms thick it will be a *Room Cluster Divide* region and a *Hall Divide* region otherwise.

Figure 3.11 shows a section of hallway containing two hall loops. The large innermost hallway was formed by a *Hall Loop* region that did not have enough space in the middle to fit rooms.

When a *Hall Loop* region is created, its placement and dimensions depend on the edges and the portals that exist in the region it is placed in. The position of an edge of the *Hall Loop* depends on the corresponding edge of the region it is placed in. If the corresponding edge is part of the building edge, the generator places the *Hall Loop* edge a distance of at least one average room away from it. If the corresponding edge is not, the *Hall Loop* edge is placed a distance of at least two average rooms away.

Once all of the edges are positioned, the generator checks the *Hallway* portals of the region it is placed in. If any of these portals are not directly across from an edge of *Hall Loop*, the generator extends the region so this will be the case. Figure 3.12 provides an example of this. The solid bordered middle region represents the *Hall*

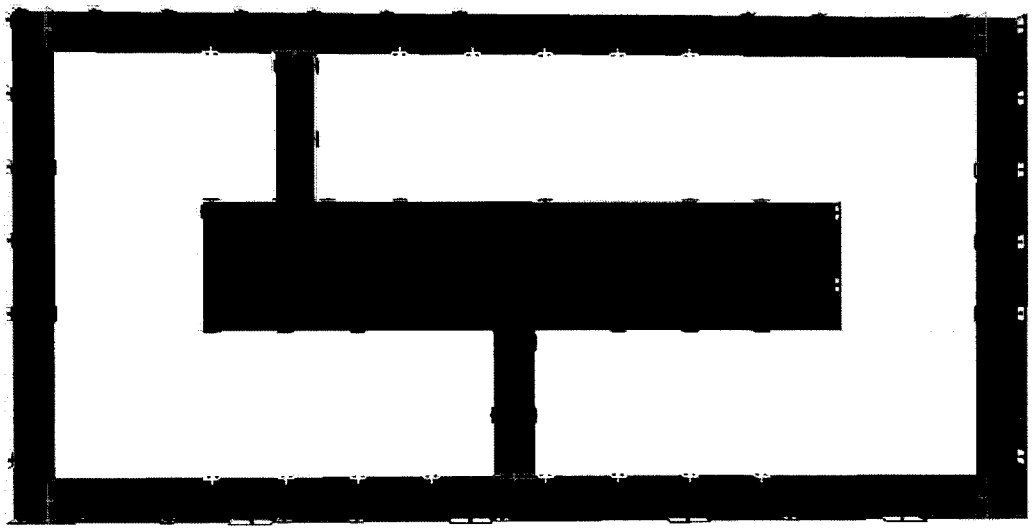


Figure 3.11: A section of a building containing two hall loops.

Loop region, and the thick black lines on the perimeter represent the *Hallway* portals. The dashed line and arrows represent where the *Hall Loop* region will be extended to.

3.8 Room Cluster Division

Room cluster division occurs in *Room Cluster Divide* temporary regions. Room cluster division divides the region into two parts, and this will create two new *Room Cluster Divide* temporary regions. Room cluster division does not place any portals between the newly created rooms, because currently we are considering all the rooms of our interiors to be private. (See rule 7 of section 3.2 for details.) The goal of room cluster division is to create an individual room for every *Room Door* portal attached to the cluster. In most cases each room will only have one door, but occasionally rooms may end up with multiple doors, since it is not always possible to place a divider between portals. Rooms are never created without doors.

The location of the divider largely depends on the location of the portals. The divider is always placed between two portals, taking care not to make it intersect with other portals in the region, and to avoid creating overly skinny regions. Division

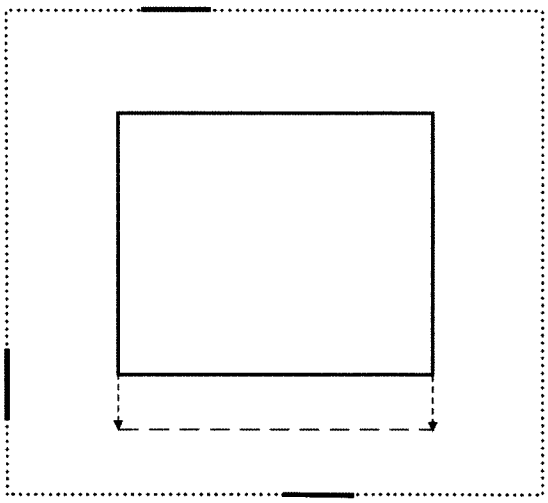


Figure 3.12: An example extending the hall loop so that every hall portal can be directly across from on of its edges

occurs parallel to either the xy-plane or the yz-plane. If it possible to place a divider parallel to either plane, the generator chooses the direction that is perpendicular to the longest side of the region. This allows for more variety in room sizes in a room cluster and discourages the creation of long skinny rooms.

Pseudo-code to divide a room cluster is provided below:

```
Given:
    DIV_INTERVAL_X = The interval between the two most extreme
                     portals along the x-axis
    DIV_INTERVAL_Z = The interval between the two most extreme
                     portals along the z-axis
    MAX_CORNER = The maximum corner defining the region's bounding box
    MIN_CORNER = The minimum corner defining the region's bounding box
    EXTENT = MAX_CORNER - MIN_CORNER
    MIN_ROOM_DIM = The minimum dimension allowed for a room

DIV_ALONG_Z = True if division should occur along the z axis
              False if division should occur along the x axis
```

```
if (it is possible to divide along the both axes without creating
    a room with a minimum dimension less than MIN_ROOM_DIM){
    if(EXTENT.X = EXTENT.Z){
        DIV_ALONG_Z = a random value of either true or false
    }
    else if(EXTENT.X > EXTENT.Z){
        DIV_ALONG_Z = true
    }
    else{
        DIV_ALONG_Z = false
    }
}
if (it is possible to divide along the z axis without creating
    a room with a minimum dimension less than MIN_ROOM_DIM){
    DIV_ALONG_Z = true
}
else if (it is possible to divide along the x axis without creating
         a room with a minimum dimension less than MIN_ROOM_DIM){
    DIV_ALONG_Z = false
}
else{
    Create a built region with the same dimensions as this region

    Replace this region with the new built region in the
    generation tree

    Return
}

DIVIDER = A divider along the appropriate axis based on
```

```
        DIV_ALONG_Z.  Initialize this divider to the middle of
        the region.

if (DIVIDER is outside of DIV_INTERVAL_X or DIV_INTERVAL_Z)){
    Move DIVIDER to the nearest location inside the interval
}

if(DIVIDER intersects a portal){
    Move DIVIDER to the closest location that doesn't intersect a
    portal
}

Divide the region based on DIVIDER

GEN_NODE = an internal generation tree node based on this region.

Make the two new regions created by the division children of
GEN_NODE

Replace this region in the generation tree with GEN_NODE
```

Figure 3.13 shows an example series of room cluster division steps to completely generate a room cluster. Every region shown is a *Room Cluster Divide* temporary region.

3.9 Built Region Creation

All of the visible geometry of a building interior is created in this step. Built regions are created from temporary regions with the same dimensions. Generating the geometry is based on the axis aligned bounding box provided by the temporary region, and

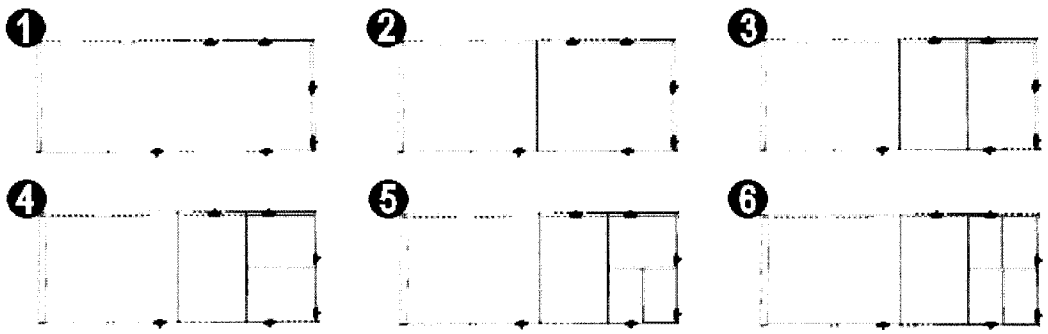


Figure 3.13: An example series of room cluster division steps. Step 1 shows the initial region, and step 2 divides this region into two parts between two of its portals. Since this can happen in both directions, the one perpendicular to the longest edge is chosen. Steps 3 through 5 go through a similar process. Generation stops at step 6 since there are no regions with more than one portal.

the portals that are attached to it. The polygons making up the walls are placed a constant thickness away from the actual boundaries of the bounding box. This gives volume to the walls of the building.

Frames are also created around every portal attached to the region. The polygons of the frames extend from the portals to meet up with the wall polygons. The geometry for door frames is typically identical for each door, so a future optimization would be to instance a master copy of a door frame for each occurrence of a door. This would greatly reduce the required memory and generation time of the building since buildings normally use the same style of door frame for every door it contains.

To add more variety in the rooms of our buildings we pseudorandomly create colors for the walls and the floors. We do not attempt to match the colors and make them look good together. When the room is rendered, these colors will be blended with the textures chosen for the room.

After a built region is created, it is populated with objects like furniture and other objects needed by the application. Object placement is described in the next section.

3.10 Object Placement

Adding objects to the building will make the generated buildings much more detailed and interesting. This is the point where objects like furniture, pillars, banisters, and other needed objects will be placed.

The object placement in our implementation so far is trivial. We simply place objects in random locations with no regard for whether they are overlapping or not. The main purpose for this is to test persistence management with generated objects. (See section 5.3.3 for details.)

We are leaving realistic object placement for future work, but we have found a promising direction to take when doing so:

Offices, bedrooms, kitchens, and other types of rooms all require different types of objects, so to place the proper objects in each room we need to know what type of room it is. However, right now in our implementation each room is essentially the same. It will be necessary in the future to assign types to each room in a semi-realistic fashion.

Once we have a technique to assign types to each room, we will need a way to quickly arrange objects in rooms in a believable way. For this purpose, we have found a masters thesis by Kari Kjolass [16]. This thesis deals with placing furniture in a valid and functional arrangement, and it seems like it would be fast enough for our needs.

One last thing to note about object placement in our buildings is that objects must be placed completely within the boundaries of the region that generates it. This means, for instance, that regions cannot place objects that straddle doorway portals. If it is necessary to place objects that straddle a single portal, it would be best for the portal itself to generate the object. Since the portal will belong to both of the regions involved, we can guarantee that the object will get generated at the proper time no matter what region is visited first. This type of object placement would be very useful for placing door objects.

3.11 Random Number Generation

Variation is achieved by the generator by using a pseudorandom number generator. These will always give the same sequence of numbers for a given seed. Each region in our building always uses the same seed, so they always obtain the same sequence of numbers. As long as the steps to generate the contents of a region happen in the same order, the result will always be identical.

To calculate the seed for a region, the midpoint and a global building seed are used. The three coordinates of the midpoint are brought to the nearest integer and then all four numbers are mixed together into a single integer. Our implementation uses the bitwise exclusive-or operation to mix the numbers.

Using all three components of the midpoint to calculate the seed causes every floor of the building to be generated differently. This provides more variety in our buildings and makes them more interesting to explore. It is common in real buildings, however, to have nearly identical layouts for every floor. Our generator can be modified to do this by simply changing the seed calculations. To generate the identical floor layouts, change the seed calculations in temporary regions to combine only the x and z components of the midpoint. The seed calculations for built regions should remain the same so that identical room contents will not be generated on every floor.

Our method to calculate seeds is very similar to the one found in [10]. They however, use Thomas Wang's 32 bit mixing function [31] to help mix the numbers. We have experimented with using this function in our implementation, but found no noticeable difference in our generation results.

A region's midpoint is used to calculate its seed. Midpoints are ideal for seed calculations because they are unique to each region, consistent for each region, and easy to calculate. An important implementation note regarding midpoints is that floating point errors may cause a region's midpoint to be slightly different each time it is generated. Our implementation rounds the components of each midpoint to the nearest integer to avoid midpoint inconsistencies.

Since the seed for each region is calculated using consistent data from each region, entire static building interiors can be generated and regenerated consistently without

storing anything but the starting parameters.

One of the first things our implementation does when generation occurs in a region is seed the random number generator. This ensures that a consistent sequence is provided if any random numbers are needed. The usage of the sequence depends on the algorithms used. The following list summarizes the algorithms currently using pseudorandom numbers in our generator:

- **Texture Selection:** Predetermined lists exist for the ceilings, walls, and floors of the room sand hallways in our interiors. The generator pseudorandomly chooses from these lists to select the textures to be used in the interior when a *Building* type temporary region generates.

- **Elevator Shaft Placement:** When a *Building* type region generates, an elevator shaft will get created. The placement of the shaft is calculated pseudorandomly so that its distance from every edge of the building region is greater than one average room thick.

- **Creating Hall Segments:** Hall segments may get created when *Hall Divide* regions generate. Hall segment creation is almost always based on an existing portal into the parent region. This portal is selected pseudorandomly from a list of possible portals. Later, if an end of the a hall segment is not fixed to an edge of the parent region it will be pseudorandomly placed a distance between one and three average rooms thick from its corresponding edge of the parent region.

- **Portal Placement:** Our current portal placement algorithm maintains an interval list to keep track of where portals can be placed. The algorithm takes constant distance steps through each interval, and at places portals according to a specified pseudorandom probability at each step.

- **Room Cluster Division:** Room cluster division occurs between two portals. If division can occur along both the x and z axes, and the length and width of the room cluster is equal, the axis to divide along will be chosen pseudorandomly.

- **Wall and Floor coloring:** When built region rooms are created, the generator randomly colors the walls and floors. This is done by generating pseudorandom numbers between a specified range for the RGB components of each color.

- **Object Placement:** After a built region is created objects may be placed in it. To do so, the generator first randomly generates the number of objects to place. It then places the objects pseudorandomly within the walls of the region with no regard to whether objects are overlapping or not.

3.12 Graphical Summary

This section gives a summary of how all the region types relate to each other. Although the actual regions of the generator are three dimensional, two dimensional regions are shown below to simplify the presentation. The *Floor divide* and *Elevator Shaft* figures are shown using a side view, and the rest are displayed using a top-down view. All of the region types are temporary regions except the *Built Regions* (BR).

Table 3.1: Legend of Region Types

Symbol	Region Type
B:	Building
F:	Floor Divide
H:	Hall Divide
R:	Room Cluster Divide
L:	Hall Loop
S:	Hall Segment
E:	Elevator Shaft
BR:	Built Region

Note: Commas are used to indicate alternative region types.

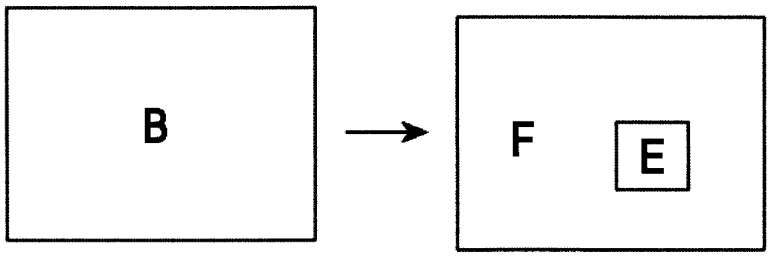


Figure 3.14: A *Building* region generates a *Floor Divide* region and an *Elevator Shaft* region.

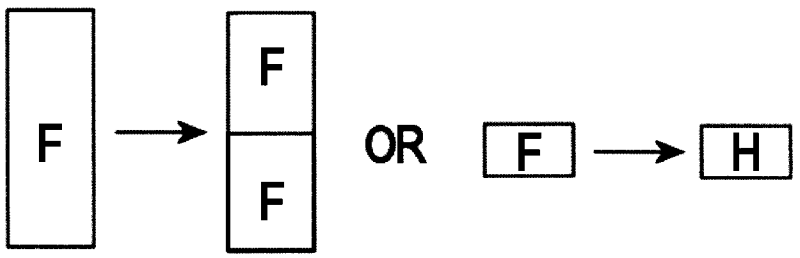


Figure 3.15: A *Floor Divide* region splits into smaller *Floor Divide* regions or becomes a *Hall Divide* region

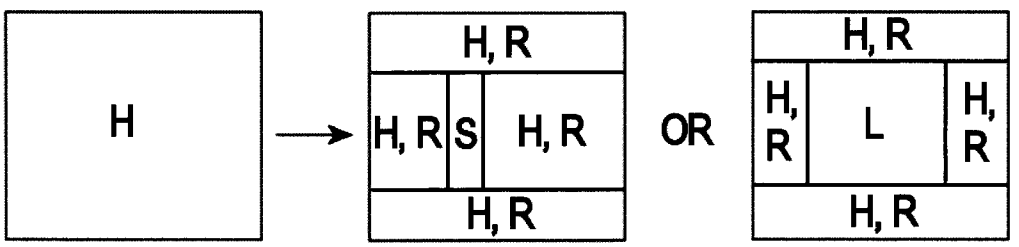


Figure 3.16: A *Hall Divide* region splits around either a *Hall Segment* or a *Hall Loop* region. Note that the surrounding regions may or may not get created depending on the positioning of the *Hall Segment* or *Hall Loop*.

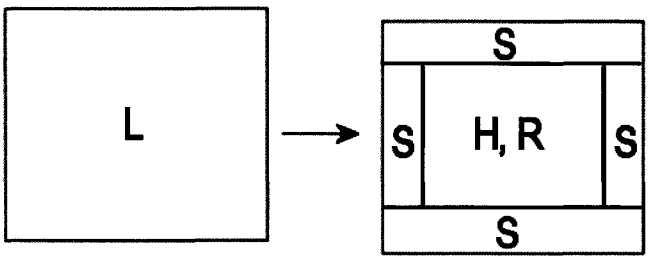


Figure 3.17: A *Hall Loop* region divides into a middle region surrounded by *Hall Segment* regions.

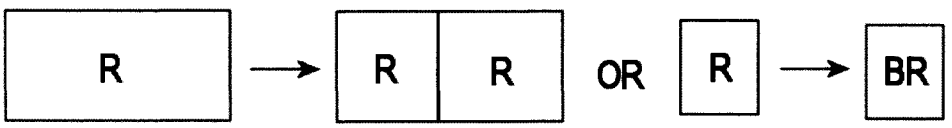


Figure 3.18: A *Room Cluster Divide* region either splits into two smaller *Room Cluster Divide* regions or becomes a built region.

Figure 3.19: A *Hall Segment* region becomes a built region.

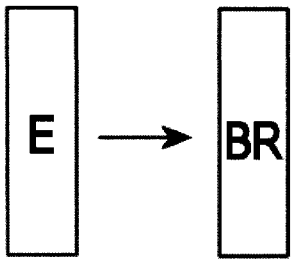


Figure 3.20: An *Elevator Shaft* region becomes a built region.

The next figures show the portal types and the types of regions that they are placed between. *Entrance* and *Window* portals are not included here, because they are not placed by our generator.

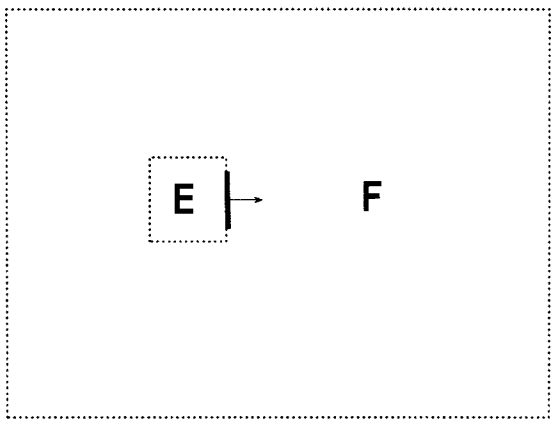


Figure 3.21: *Elevator* portals are placed between *Elevator Shaft* and *Floor Divide* regions.

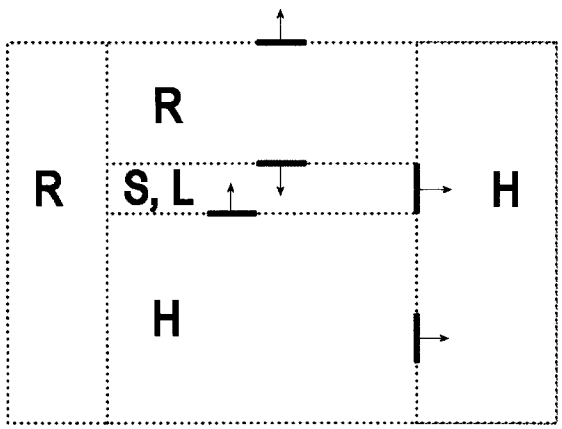


Figure 3.22: *Hallway* portals are normally placed between hall and *Hall Divide* regions, or two *Hall Divide* Regions. There is also a situation where *Hallway* portals are placed between hall and *Room Cluster* regions. This occurs when an existing *Hallway* portal is found directly across a hall region over a *Room Cluster* region

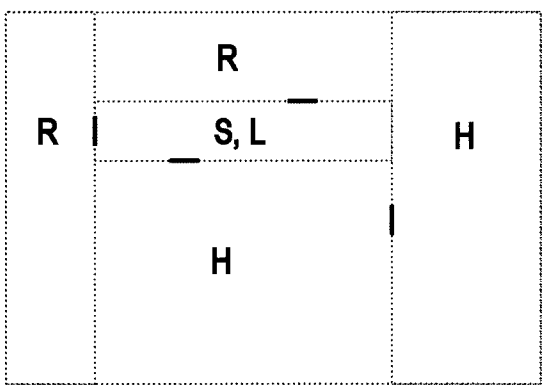


Figure 3.23: *Room Door* portals are placed between hall and *Room Cluster* regions, hall and *Hall Divide* regions, or two *Hall Divide* regions.

Figures 3.24 and 3.25 provide an generation example showing all of the region types. These examples were made using screen shots from our implementation. Excluding the last step, generation will always occur in the region containing the red dot. The steps are described below:

1: The initial region is a *Hall Divide region.*

2: The *Hall Divide region* splits into two new *Hall Divide regions*, two *Room Cluster Divide* regions, and one *Hall Loop* region.

3: The *Hall Loop* region divides into four *Hall Segment* regions, and one *Hall Divide* region.

4: The *Hall Divide* region divides into three new *Hall Divide* regions, one *Room Cluster Divide* region, and one *Hall Loop* region.

5: The *Hall Divide* region divides into two *Room Cluster Divide* regions, and one *Hall segment* region.

6: The *Room Cluster Divide* region divides into two new *Room Cluster Divide* regions.

7: The *Room Cluster Divide* region divides into two new *Room Cluster Divide* regions.

8: The *Room Cluster Divide* region divides into two new *Room Cluster Divide* regions.

9: The *Room Cluster Divide* region divides into two new *Room Cluster Divide* regions.

10: The *Room Cluster Divide* region turns into a Built region.

11: The building portion generated completely. All regions shown here are Built regions. All regions except the hallways and and the region that contains the point shown in wire-frame.

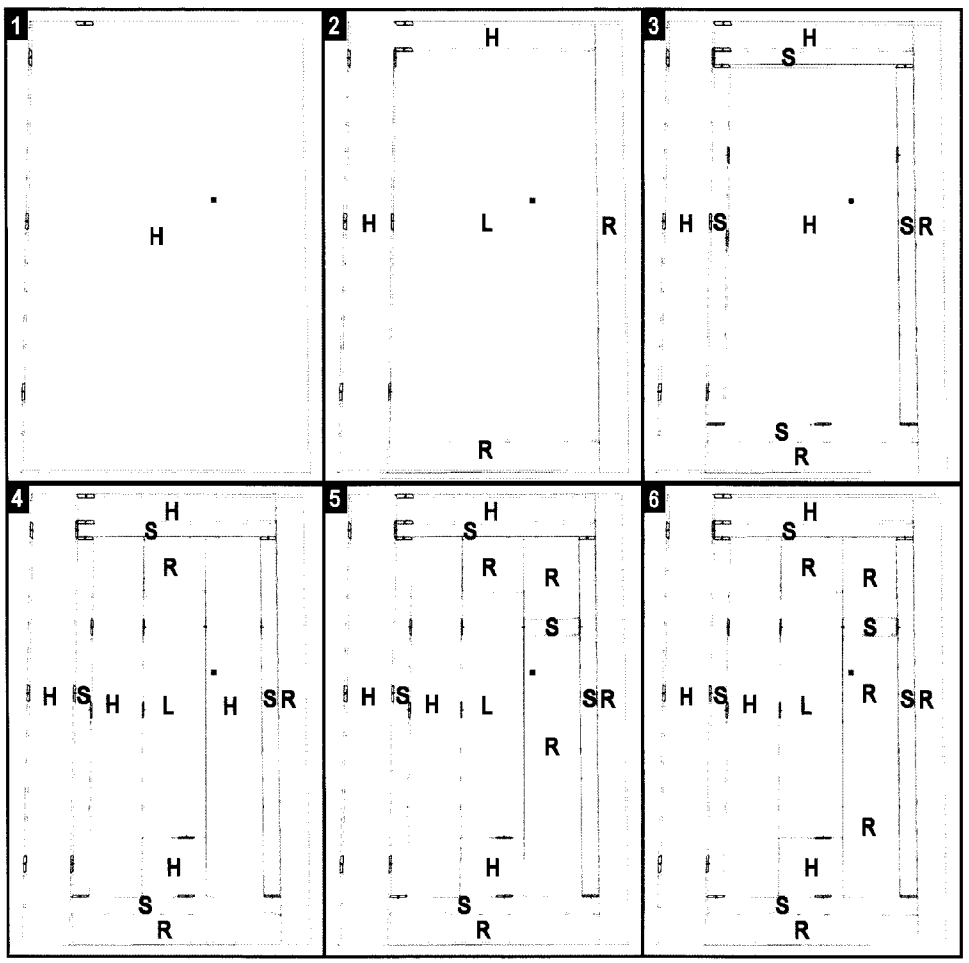


Figure 3.24: A generation example from our implementation

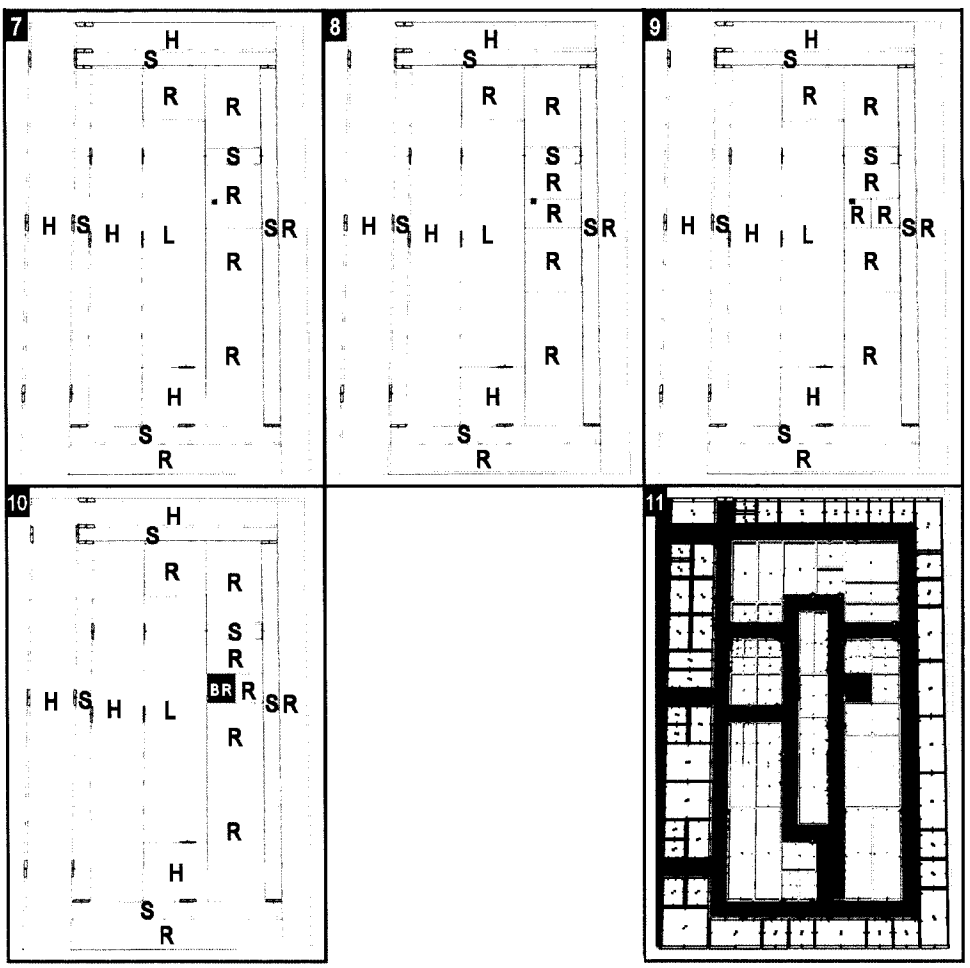


Figure 3.25: The generation example continued

Chapter 4

Memory Management

This chapter covers how we manage the memory of our generator. In order to achieve the memory savings our generator provides, the memory from unneeded regions must be released at some point. Our system frees unneeded regions from memory by reversing the generation process in unneeded portions of the building. This allows our generator to generate, reverse, and regenerate areas of the building indefinitely.

This chapter is divided into sections. Section 4.1 covers the generation tree, which is the main data structure we use to reverse generation. Section 4.3 explains how we reverse generation using the generation tree. Finally, section 4.2 covers how we cache built regions created by our generator

4.1 The Generation Tree

To manage the memory of our system, regions of the building need to be deleted. Since these regions may be needed again in the future, care must be taken to ensure that they can be regenerated consistently. Regenerating a region that is identical to its original requires using a region that is identical to its parent. Obtaining such regions can be achieved by reversing the generation process. However, reversing this process without storing any details of what has previously taken place during generation is not a trivial task. We expect that this would be of equal or greater difficulty to generating the building in the first place. Fortunately, the problem is

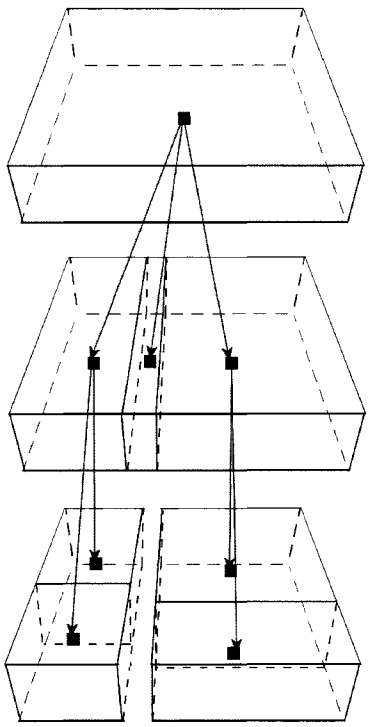


Figure 4.1: A small generation tree.

straight forward if a generation history is maintained.

Our system uses a generation tree to maintain this history. A tree data structure is ideal to maintain this history because our recursive top-down generation approach is tree-like in nature. We can also capitalize on the tree structure for needs other than reversing generation. The generation tree allows us to make efficient point location calculations, this ability greatly increases the usefulness of our generation approach in realtime applications. Point location in our system is described in detail in section A.2.)

Each internal node of the tree represents a past generation step. These nodes contain an axis aligned bounding box, a region type, and connections to its parent and one or more children. Figure 4 shows an example of a small generation tree.

Every leaf of the tree is also a region of the building. This allows quick access into the needed parts of the tree when regions are deleted and efficient point location.

The tree grows after each generation step. When a temporary region is split, an internal node is created that is given that region's bounding box and region type. Then the newly generated regions are set to be the children of the new internal node.

4.2 Caching

Whenever a built region is created, a pointer to it is stored in a least recently used cache. When the cache goes over a predetermined size we delete regions by starting with the least recently used. Note that the set cache size is not a rigid limit to how many regions are kept in the cache. Visible regions will always be kept in the cache, and it is possible to have a greater number of visible regions than the cache size. Here we are assuming that the system will have at least enough available memory to hold the visible regions of the building.

Updating the cache is quick and simple. The cache is implemented with a doubly linked list to provide constant time updates, and each built region of our buildings stores its location in the list to allow constant time access to its nodes. When a built region is used, it moves its node to the back of the list. This keeps the elements of the list in the order from least to most recently used. In addition to the constant time updates, the doubly linked list also allows nodes to be moved without changing the memory locations of the rest of the nodes. This is important because it ensures that when a node is moved all of the pointers to the other nodes in the list will not be invalidated.

A doubly linked list was also used to implement a least recently used cache in [10]. To provide access into the list they use a balanced tree. This allows list nodes to be accessed in *O(log n)* time. In our implementation however, each built region stores its location in the list. This allows access in *O(1)* time.

The least recently used cache was not the first caching method we used. Initially, we experimented with a graph-distance based cache. The graph distance between two regions is defined as the shortest number of portals required to pass through to travel between the two rooms. With this caching method, we kept in memory the union of the built regions that were visible and the built regions that were greater than or

equal to a specified graph distance from the camera's region.

When we tested our system using this cache we achieved the best memory consumption and performance when the cache distance was zero. Memory consumption increase and performance decreased as the distance was increased. This showed us that this caching method was not only ineffective, it was actually hurting our results. This lead us to adopt the least recently used approach that was presented at the beginning of this section. This new approach turned out to be much simpler to implement and gave better results.

4.3 Deleting Regions

Our cache keeps track of all the built regions in a building, and we control the memory in our system by deleting built regions. After a built region is deleted, we then move up the generation tree and remove the temporary regions and internal nodes that are possible to remove. The exact steps to delete a built region can be found below:

1. All of the changes for the built region are saved into the persistence manager. (See chapter 5 for details.)

2. A temporary region identical to the one that generated the built region is created.

3. The new temporary region then takes the place of the built region in the generation tree.

4. If the siblings of the new temporary region are all temporary regions as well, they will all be merged into a single temporary region and replace their parent. This step will then repeat for the newly merged temporary region.

Merging temporary regions is essentially reversing the generation step. This makes merging simple because most of the information needed is provided by the generation tree. The parent of the merging regions will already contain the axis aligned bounding box and the temporary region type needed to create the merged temporary region.

The remainder of the work, is processing the portals of the regions. This is done by examining all of the portals of the merging temporary regions. Any portal that connects two of the merging regions is deleted, and the rest are kept and connected to the merged region.

Chapter 5

Persistent Change Management

The generation process presented so far will provide a consistent environment unless the building or its contents are changed after generation. Since the changes are not the result of the generation process, they would disappear if the altered parts of the building are regenerated. In order to have consistency in a dynamic environment, the changes made to the building must be stored. Our system can handle changes made to built regions and portals.

All changes made to a region will be stored in a single record that is accessed through a hash map. A hash map was chosen to store the needed changes to the building to provide quick look up, insertion, and deletion. These operations must be as fast as possible in order to achieve realtime speeds during generation. Records will be described in detail in section 5.1. Changes made to portals are placed in a separate hash map from regions. See section 5.3.4 for details.

The midpoints of the regions are used as the keys into the hash map. The midpoint of the region makes an effective key because it can be quickly calculated, and it is easy to keep unique. Errors may occur if non-unique keys were produced, since a region may obtain a record meant for a completely different region when generating. Using the midpoint as the key, there are two cases where keys may not be unique and both can be easily avoided by the generator:

- **When regions are overlapping or contained in each other.** This case will not happen if generation rules 1 and 2 are followed. (See section 3.2)

- **When two or more midpoints are so close together that precision errors occur.** For this to occur, the regions would be so small that they would be effectively unusable. Our generator does not produce regions this small.

Changes made to regions are not immediately stored in the hash map. They will be stored later, when the region that contains them is deleted. This approach resembles write back caching in computer architecture[32]. It can greatly decrease the number of updates needed to the hash map, since the only the state of the changes when the region is deleted will be stored. This is especially important if there will be moving objects in the application since these objects change their state every frame that they're in motion.

To apply the changes during generation to a region, the region is first generated normally. Then the key for the region is generated, and used to search the hash map. If the record is found, the changes will be applied to the region.

5.1 Records

Three different types of changes are possible in a region. We chose to store all of these changes in a single record. To handle all three changes, region records contain two collections. The first collection holds either non-generated objects or color changes. The second holds the indices of objects that should not be generated in that region.

Portal records are different records than region records. These records hold the changeable portions of a portal's data. Portals in our implementation can either be completely open or completely closed, and this is all that can change. To handle this, our portal records store a boolean variable to indicate the position it is in.

5.2 Removal Time

Records are given a removal time when they are inserted or updated in the hash map. This is done to support temporary changes, and to reduce the number of records stored. To implement this, whenever a record is inserted into the hash map,

the time that the record will be removed is calculated. Then an element containing the key and the removal time is inserted into a heap. The top element of the heap will always contain the key of the first record that needs to be removed. If the top element does not need to be removed, nothing needs to be removed.

When records are updated the corresponding element in the heap must be given the new removal time. To update the removal time of the record, a linear search is executed through the heap in order to find the element. The element is then given the new removal time and the heap is updated.

5.3 Types of Change

Our system allows four different types of changes: The appearance of faces, non-generated objects, generated objects, and portals can be all changed. The details of these changes can be found below.

5.3.1 Face Appearance Changes

The ability to change the color and texturing of the faces in the region is useful for developers wishing to fine tune the building's appearance. Our implementation has the ability to change the color of the faces for all of the faces in the room. With this implementation each face of the region is given a color which is blended with its texture.

When a color change is applied to a region, the color is assigned to all of the faces in it, and it is also stored separately in the region. Storing the color is done to avoid calculating this color when it is time to save, and its presence also indicates if a color change needs to be saved at all. Later, if a color is stored in the region when it is time to save, it will be saved in the region's record.

5.3.2 Non-Generated Object Changes

Any object that will not be placed by the generator is considered a non-generated object. These objects can be placed by designers before the program is used, or placed

during runtime, either by a user or another part of the program.

Placing or altering this type of object is fairly simple. Placement only requires putting an object in the destination region's object list, and altering any non-generated object only requires changing that object's state.

Later, when the region saves its changes, all non-generated objects will be saved to the region's record. To regenerate the region, these objects simply need to be copied back to the region from its record.

When a region saves its changes our system currently rewrites its record. This simplifies the removal of non-generated objects because the system does need to update any records each time an object is removed.

5.3.3 Generated Object Changes

Any object placed by the generator is considered to be a generated object. These objects can be regenerated consistently without saving changes if they are left unchanged from how the generator made them.

Generated objects are kept in a separate location from the non-generated objects. When a generated object is initially changed, it is moved to the location of the non-generated objects, and a placeholder object is put in its place. After this, the changed generated object will be treated exactly the same as any other non-generated objects.

When the region saves its changes, it will loop through its generated objects. Whenever a placeholder object is encountered the index of that object will be placed at the end of a list in the region's record. This new list will be referred to as the don't-generate list. It contains the index of every object that should not be regenerated in the future. This is needed to avoid creating duplicate objects.

It is important to note that the don't-generate list relies on the fact that the generated objects always appear in the same order. If this were not true, the indices in the don't-generate list would become invalid.

Changes are restored using the same loop that generates the object. This ensures that all of the objects are processed in the same order. If the index of an object is in the don't-generate list the object will not be created. A placeholder object will be put

in its place to preserve the indices of the remaining objects, and the pseudorandom number generator will be advanced to the state needed to generate the next object correctly. In our implementation this requires generating the same amount of random numbers that are needed to create the actual object.

5.3.4 Portal Changes

Portal changes such as opening and closing doors can also be handled. However, providing persistent changes for portals is different from the previous persistence solutions because portals always belong to two regions, whereas the changes in the previous solutions were always confined to a single region. Also, the lifespan of portals can be different from the lifespan of the regions they are attached to. This is because portals often change regions during the generation process when regions are split or joined together. For these reasons, storing portal changes in region records is not the best direction to take.

To verify this, we experimented with solutions that stored portal changes in region records. When the time came to save the changes of a region, we stored the state of any portal that wasn't at its initial state in the regions record. This however, led to some problems.

First, the changes for portals were saved and applied at unnecessary times. Mainly, the only time it is necessary to save changes for a portal is when the portal is deleted, and the only time it is necessary to apply portal changes is when that portal is created. Since regions get created and destroyed a lot more than most portals, a lot of unnecessary time and memory was used for portal changes.

A second problem was that information for a single portal would be found in multiple region records, and the portal itself. Not only is this inefficient in memory, it led to consistency problems. There were situations where the state of the portal in one region's record would become different from the record in its neighboring region.

Because of these problems we took a different approach. Instead of storing portal changes in region records, we made a separate hash map and records made specifically for portal changes. This not only eliminated the problems of the other solutions, it

turned out to be simpler and more flexible than them as well.

Handling persistence for portals is virtually the same as for regions. Just like regions, the midpoint of the portal is used as the key in the hash map. Changes are only stored when a portal is deleted, and only applied when portals are created. Removal of records at an appropriate time is even handled the same. The only real difference is that a different type of record is stored in the hash map. Because of this, very little extra code is needed to handle persistence for regions and portals separately.

5.4 Objects Occupying multiple regions

A problem arises when objects can occupy multiple regions. All of the regions that an object occupies must know of the object so that it will always get rendered when it should be, and care must be taken so that the object will get placed and rendered properly no matter what order the occupied regions are deleted or generated.

We handled this problem with the following method: when the changes for a region are saved, store a pointer to the object in the region's record and then to increment a counter on the object that counts how many records refer to it. When records are deleted later this counter will be decremented, and if the counter reaches zero, the object can be deleted.

When an object moves, the system must make sure that all of the space that the object occupies is generated. This will avoid collision detection errors, and cases where objects get moved out of the spaces of deleted regions that hold records for them.

5.5 Discussion

This chapter presented persistent change management for our generated buildings. This allows the system to delete regions without destroying the changes that users have made. It also allows developers to make changes to built regions, like texture changes and object placements.

It is important to note that the manager presented here can only handle changes made to built regions, and portals. This should be powerful enough to handle all of the possible changes that users can make because built regions and portals are all users can interact with.

Developers on the other hand, may want to alter temporary regions as well. This would give developers extensive control over the generated results, as it would allow them for example, to move walls, place or remove portals, and even change the seeds of temporary regions.

The persistence manager would have to be extended for such changes, because temporary regions of different generations may overlap each other. With our current manager overlapping regions may cause hash key conflicts so they must be avoided. These conflicts can be avoided by using more information in the key. For instance, conflicts can be avoided if the two corners that define the region's bounding box and the region's type are used in the key. Such an extension to our persistence manager is left for future work.

Chapter 6

Results

Our system was implemented in C++ using OpenGL and GLUT [15]. Our testing machine had a 3 Ghz Intel Pentium 4 CPU, 1 GB of RAM, and an ATI Radeon 9800 pro graphics card with 128 MB of RAM. The operating system was Windows XP. Testing was conducted using an 800x600 OpenGL window in 32 bit color.

We tested the system as a whole by automating walks through the building. Rendering and collision detection were enabled during the test, and all doors were set to be open. Each walk visited all regions of its building in a depth first search order. To create the graphs shown below, we tested five different seeds for each building size and then averaged the results.

To test the persistence system, the tests changed the color of every room that was visited, and each change was given the maximum possible lifetime to ensure that the changes will last the duration of the test. Since every region of a building will be visited by the end of the test, a record will exist for every region in the building by the end. This provides a worst case scenario to observe.

We tested two extreme cases in our system. In the first case, the system deleted the maximum number of regions possible at any given time. This restricted the cache to exclusively contain visible regions, and provided a worst case scenario in terms of speed because each newly visible region must be generated. The cache however, will always be at its minimum size, which is the best case in terms of memory.

In the second case of our tests we did not delete any regions from the cache. This

was meant to provide a best case scenario in terms of speed because the regions of the buildings would never need to be generated more than once. This also provided a worst case scenario in terms of memory because the entire buildings were stored in memory at the end of each test.

Figures 6.1, 6.3, and 6.4 compare the results of the two test cases. The remainder of the graphs use data from the first test case exclusively.

To indicate the size of a generated building, we use the total number of regions in a building if it were generated completely. Regions in our buildings will either be individual rooms or straight sections of hallway. The building shown in figure 1 had 14537 regions and 100 floors.

The performance of our system is high even though we are generating the building during realtime. Figure 6.1 shows the average frame rates achieved during the test. For all of the buildings we tested, the average frame rate exceeded 300 frames per second. These frame rates show that it is possible to add more complex environments and and other calculations and still achieve a decent frame rate.

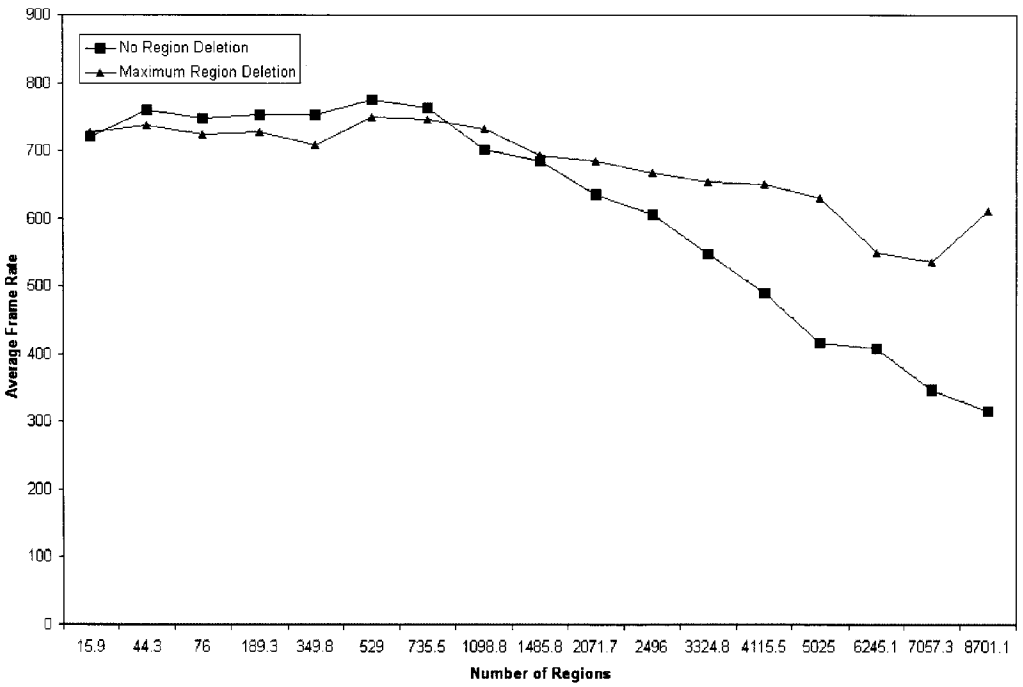
Our system achieved these frame rates even with larger buildings made up of roughly 1.3 million polygons. Figure 6.2 shows the average polygon counts of the completely generated buildings that we tested. The polygon counts of this figure exclude the polygons of the building's randomly placed objects.

Between the two test cases, the second case achieved slightly higher frame rates for smaller buildings, but the frame rates dropped much quicker than the first as the buildings got larger. For larger buildings, the first case had much higher frame rates. This was especially true for the the largest buildings where the frame rate approached almost double that of the second case. This result was unexpected. We predicted that the second case would have better frame rates since it never needs to regenerate parts of the building. We do not know the cause of this performance drop, but believe that it could have been caused by swapping due to the large increase of memory use in case two.

Similar to the frame rates of our system, the generation times of our generator are low enough to allow further processing each frame and still allow a decent consistent frame rate. Figure 6.3 shows the average times our generator took to generate around

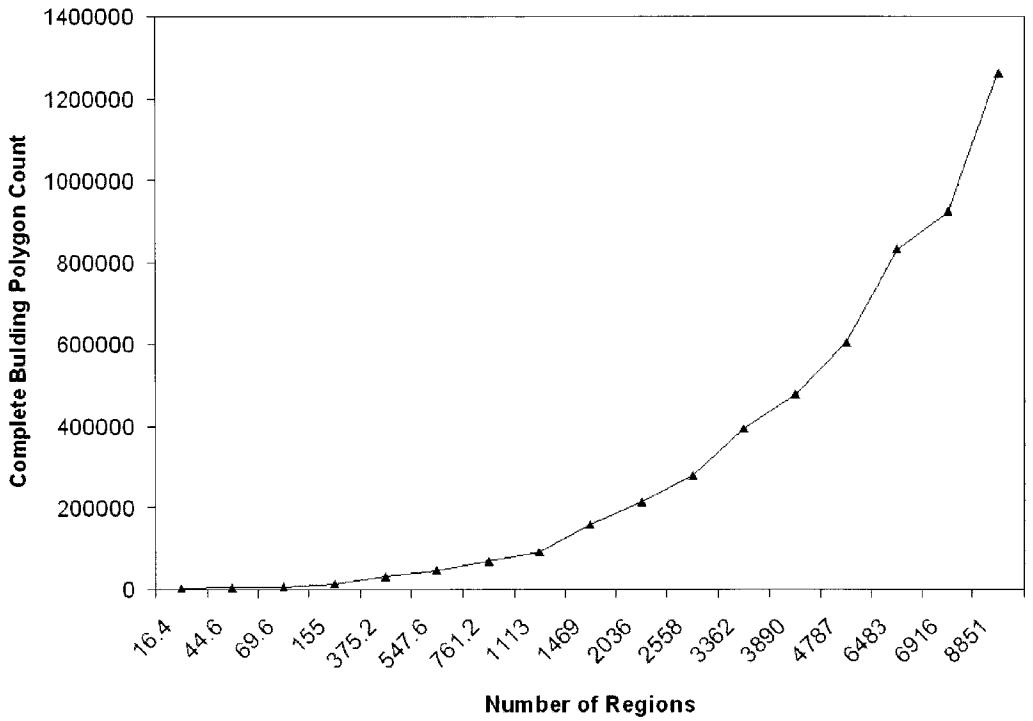
points. For all of the buildings we tested the average times it took to generate around points were below 4.5 milliseconds. Between the two test cases, the generation times were significantly higher in the first case for most of the buildings.

Figure 6.1: Frames per second vs the number of regions in a complete building



The second case of our tests used more memory than needed for the entire buildings, because the generation tree can add significant overhead if every region is kept in the cache. The second case however, deleted as many regions as possible and achieved huge memory savings as a result. Figure 6.4 gives a comparison of the memory required to hold the entire building with the maximum memory we needed for both test cases. The savings in the first case are modest for smaller buildings, but quickly increase as buildings get larger. In buildings with an average of 155 regions, the average maximum amount of memory we required was just over 16 percent of the required memory of the entire building. The largest buildings we tested had an average of 8851 regions and required an average of just under 42 MB to store entirely.

Figure 6.2: The polygon count of a complete building vs the number of regions in a complete building



Using lazy generation, we only needed on average 2 MB to explore the buildings entirely. This is just under 5 percent of the memory for the total building. Although a single 42 MB building would not be expensive to keep entirely in memory, a city full of these buildings would be expensive to store even on permanent storage like a hard disk or DVD.

We require even less memory to store the buildings off-line. To completely regenerate our buildings, we require the starting parameters and the persistence data. If a lot of changes were made, the persistence data will make up the majority of space needed to store and regenerate the buildings. For all of our tests, the persistent data fluctuated between 3.5 and 4 percent of the memory needed for the entire building. This allows huge memory savings when permanently storing the building. Figure 6.5 shows the maximum memory used by the persistence records as a percent of the memory needed for the entire building.

Figure 6.3: The average time taken to generate around a point vs the number of regions in a complete building

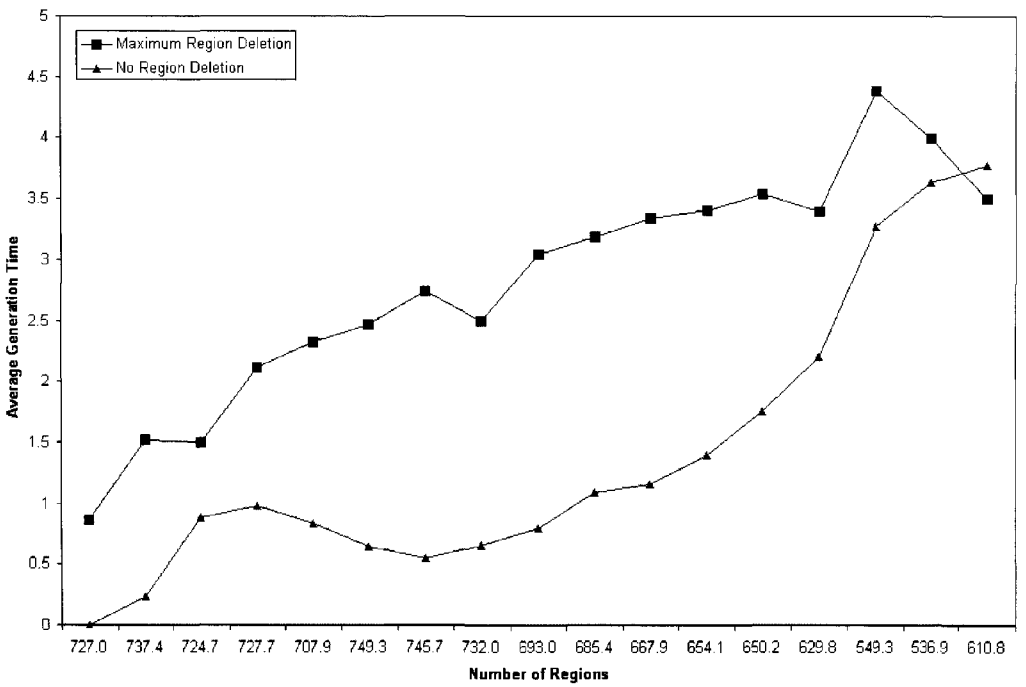


Figure 6.4: A comparison of the total memory needed for an entire building to the maximum memory used by our generator.

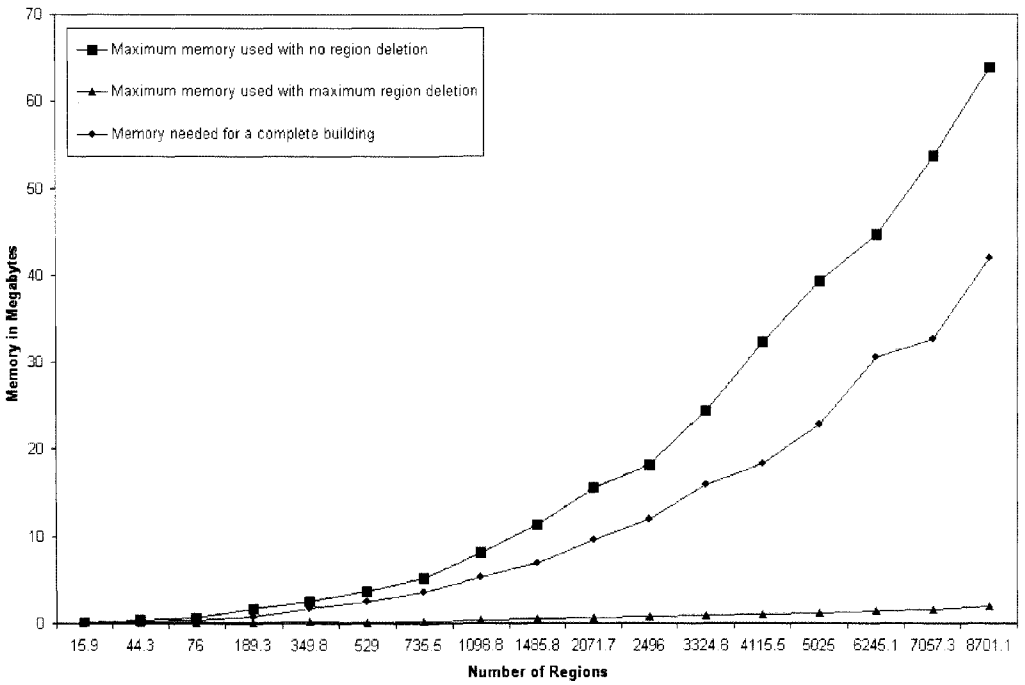
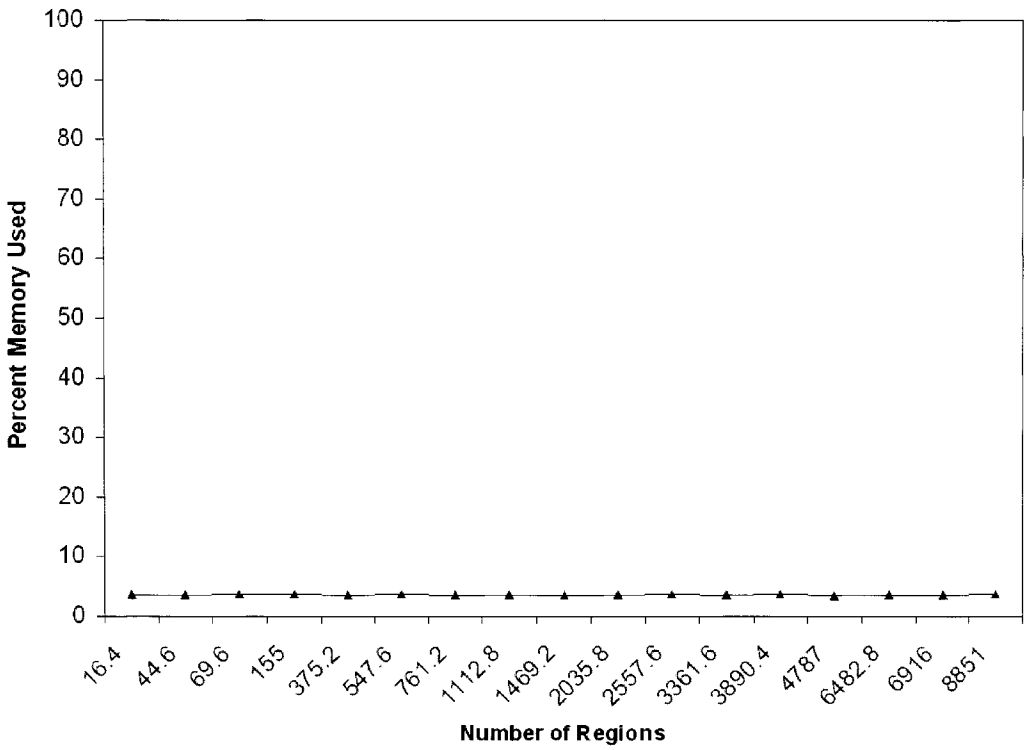


Figure 6.5: The maximum memory used for persistence as a percent of the memory needed for a total building vs the number of regions in a complete building



Chapter 7

Conclusions and Future Work

An approach to generate building interiors in realtime has been presented. With this approach, we can significantly increase the amount of content in a virtual world, without greatly increasing creation time, required memory, and loading times. Also, despite the fact that the building is generated on the fly, our system achieves realtime frame rates even when large buildings are generated.

Our system generates building interiors in a lazy fashion that generates regions of the building only as needed. Coupled with our memory manager, our system can provide huge memory savings in comparison to a completely generated, uncompressed building.

Our generated interiors are interactive, allowing dynamic changes to be made by its users. Even though significant portions of the building can deleted during runtime, these changes are kept consistent through the use of our persistent change manager. The manager also makes pre-release developer changes possible. This allows building interiors to be rapidly created without sacrificing total control over the results. We are not aware of any other realtime generated environments that provide persistent change management.

The persistence manager can also be used to overcome a major of the limitation of our generator. In order for our system to be useable in most real applications the generated buildings will require realistic object placement. This will make the buildings much more believable and interesting. Although our system does not have

the capability of automatically placing objects realistically, it still provides the means to do so externally. Through the use of our persistence system, objects can be placed in the building either manually or through the use of a separate object placement application. We know of two existing approaches to do so: One approach is presented in the master's thesis of Kari Kjolass[16], and the other was developed by Larive et al.[17].

We are attempting a difficult feat with this generator. Great time and care is put into designing real buildings, but we are generating our buildings in realtime. For our system to achieve more than 30 frames per second, the system has less than 33 milliseconds for all of its calculations per frame, and only part of this time can be used to generate the visible portions of the building. This is such a short amount of time that the generator is essentially trying to generate something with an intelligent design without taking any time to design it.

With this in mind, we are not attempting to make the generator compete with the designs of real buildings. Our generator is largely intended for use in video games. For these applications, the generator does not have to produce perfectly realistic results. It simply has to provide believable and interesting interiors to explore.

We believe that our generator does generate believable enough interiors, but major improvements are still needed before the generator will be ready for commercial applications like video games. Luckily, the frame rates and generation times are quite good, and thus make it possible to add major improvements and still achieve decent, steady frame rates. Specific improvements and other areas of future work are discussed in the next section.

7.0.1 Future Work

There are many opportunities to expand this work further. Much of the work in future will involve generating more complex and interesting buildings, as well as creating different types of buildings. Work should also be done to take the generator beyond creating room structure. Meaningful rooms should be created with realistic contents. For this purpose a quick method will be needed to assign different room types to

regions in a meaningful manner, and to place furniture in these rooms in a valid arrangement.

The addition of realistic lighting will also greatly enhance the appearance of the generated buildings. This however, will create new challenges since regions that shine lights into visible regions will have to be generated even if they are not visible themselves.

These additions would make the generator significantly better, but an unfortunate possibility with realtime generated buildings is that they may never be able to compete with the quality of traditional human-made content. Hybrid approaches of mixing human-made content with generated content may allow applications to get the best of both worlds. For instance, one possible hybrid approach may be to manually model the important parts of a rough building, and then place temporary regions in the remainder of the space. This would allow the artists to focus their attention on modelling the important, and hard to generate aspects of a building, and then let the generator take care of creating the more monotonous sections of the building. An example of this would be to model a dramatic lobby for a building, and then let the generator take over to create floors of offices or apartments. Creating buildings in this fashion would allow users to explore entire, realistically sized buildings, put more control into the hands of designers, and allow for more interesting and unique building interiors.

It would also be worthwhile to explore the combination of realtime procedural environments with destructible environments. Destructible environments can give users the freedom to break barriers and visit locations that may not otherwise be accessible or even visible. Content will be needed for the opposite sides of these barriers, and providing this content through the use of our generator would have many benefits. It would allow designers to provide much more freedom in what users can destroy and explore, without greatly increasing the required memory, development time, and loading times. Furthermore, excessive damage in a destructible environment can cause an excessive number of regions to be visible at the same time. This has the potential to slow an application down to a halt if enough regions become visible at once. With an extended version of our generator, damage could be repaired easily.

To do so, all our system would need to do is regenerate the affected regions. This would restore the geometry of the regions to their initial generated state.

Finally, if this generator was combined with other generators, larger, more complex environments could be produced. For instance, if this generator was combined with a realtime city generator, cities with complete buildings could be explored. If these cities could be placed within a realtime generated terrain, entire countries may be explored. It would be interesting to know just how large and complex an environment is possible through combining generators. Perhaps it will be possible to create and explore almost infinite universes much like our own.

Appendix A

Important System Calculations

There are a number of important calculations that our system carries out in order to function that are unrelated to building interior generation and persistence. Our system requires efficient collision detection, point location, and visibility determination in order to function and perform decently. These calculations are described in the next sections

A.1 Collision Detection

Collision detection is essential for most applications with interior environments since interaction it is nearly impossible without it. Our system uses a simple and efficient collision detection scheme based on the region-portal graphs created by our generator.

An object can only collide with other objects that share one or more of the same region. Using this fact, we can quickly eliminate most of the building from consideration when detecting collisions. This is done in constant time for an object if the system already knows what regions it is in. Our system tracks what regions moving objects are in to make this possible.

Tracking is made easy through the use of a building's region-portal graph, assuming that the initial region that contains an object is known. The system will know when it moves into another region when the object intersects or crosses a portal of this region. When this occurs, the moving object will update a list of the regions it

occupies.

There are some types of motion that the system cannot track, such as moving through walls or teleporting. If this is the case, or if the initial region that contains the object is unknown, the system can still quickly determine the containing region, using the point locator provided by our system. This is described in section A.2. Our point locator will still work even if the containing built regions do not initially exist.

Once the regions that an object occupies are known, the system simply tests for each face and object for collisions. This is done in linear time for each moving object.

A.2 Point Location

Our system has two types of point location: The first simply returns the immediately existing region that contains the point. This region can either be a temporary region or a built region.

The second type extends the first so that a built region will always be returned. If the point under consideration is contained in a built region, it will return that region. However, if is is contained within a temporary region, generation will be triggered within this region around the point. The newly generated built region that contains the point will then be returned.

Our system requires efficient point location to carry out many of its tasks. One such task is to calculate the regions that contain an object when it is teleported or moved through walls. It is also used to calculate the containing regions for an object for collision detection purposes, if they are not already known. Another, less obvious use, is to use points to refer to regions that may or may not exist in the future. Our system has to be very careful with pointers to regions because the regions may get deleted in the future. Our second type of point location provides an easy and safe way to refer to regions, because this type will generate the required region if needed.

For both types of point location we require an efficient way to find the immediately existing region that a point lies in. Our memory manager creates and maintains a data structure that can be used for this purpose: The generation tree we described in section 4.1 has a secondary purpose as a data structure for efficient point location.

This is made possible because the bounding boxes needed at each node of the tree form an axis aligned bounding box tree (AABB tree)[5].

Using the tree, a simple recursive algorithm can be executed to find the region that the point lies in. The algorithm starts at the root of the tree. It then examines the children and traverses to the first child it finds with a bounding box that contains the point. This is repeated for each non-leaf node. When a leaf is reached, the algorithm will halt and return the region found. The traversal to the bottom occurs in logarithmic time.

From this point, the point locator will either return the region found, or generate and return the built region that contains the point, depending on what type of point location is used.

A.3 Visibility

We are using portal-region graphs for efficient visibility determination[20]. This is an essential part of our lazy generation method, and is required to render large interiors at a decent frame rate. The use of portals for visibility determinations is common for interior environments, because of the abundance of walls that obstruct the view, and because they can be divided into regions and portals easily.

We render the visible regions of our interiors using a simple recursive algorithm. The algorithm starts with the region that contains the camera. If this region is not known, our point locator can be used to quickly find this region. (See section A.2.) Pseudo-code for the algorithm can be found below:

```
R = the region containing the camera V = the camera's view volume

RenderRegions(R, V){
    for (every portal of R that intersects the view volume){
        O = the region on the other side of the portal

        if (O is a temporary region){
            generate the built region on the other side of the portal.

            O = the new built region.
        }

        if (O has not been rendered this frame){
            N = the new visible view volume through the portal

            RenderRegions(O, N)
        }
    }

    render R
}
```

Bibliography

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, USA, 1977.

[2] H. Ammeter and M. V. Mikailyuk. Generation and Rendering of Virtual Terrain. *Computer Graphics and Geometry*, Vol. 2, No. 2. 1999

[3] M. P. Atkinson, P. J. Bailey, K. J. Chisbolm, W. P. Cockshott and R. Morrison An Approach to Persistent Programming. *The Computer Journal*, Vol. 26, No. 4, 360–365, 1983.

[4] S. Benford, C. Greenhalgh, T. Rodden, and J. Pycock. Collaborative Virtual Environments. *Communications of the ACM.* Vol. 44, No. 7, July 2001.

[5] Gino van den Bergen, G. Efficient collision detection of complex deformable models using AABB trees.*Journal of Graphics Tools*, Vol. 2, No. 4, 1-13, Jan 1998,

[6] Binary Worlds. Descensor Engine. Website, Jan 2006. http://www.binaryworlds.com/products.html.

[7] Blizzard Entertainment. *Diablo II.* Website, May 2006. http://www.blizzard.com/diablo2exp/

[8] A. L. Brown. *Persistent Object Stores*, PhD thesis, University of St. Andrews, 1989.

[9] S. C. Dollins. Modeling for the Plausible Emulation of Large Worlds. PhD Thesis, Brown University. 2002.

[10] S. Greuter, J. Parker, N. Stewart, and G. Leach. Real-time Procedural Generation of 'Pseudo Infinite' Cities, In *Proceedings of GRAPHITE 2003*, ACM SIGGRAPH, 87–94, 2003.

[11] E. Hahn, P. Bose, and A. Whitehead. Persistent Realtime Building Interior Generation. *ACM SIGGRAPH Video Game Symposium* Boston, July 2006

[12] E. Hahn, P. Bose, and A. Whitehead. Lazy Generation of Building Interiors in Realtime. *Canadian Conference on Electrical and Computer Engineering* Ontario, May 2006

[13] J. C. Hart. On Efficiently Representing Procedural Geometry. 1994.

[14] J. C. Hart. Procedural Synthesis of Geometry. *Texturing and Modeling: A Procedural Approach* Third Edition, 305–334 Morgan Kaufmann Publishers, San Fransisco. 2003.

[15] M. J. Kilgard, *OpenGL Utility Toolkit Programming Interface API.* Silicon Graphics Inc. Website, Jan 2006. http://www.opengl.org/resources/libraries/glut.html

[16] K. A. H. Kjolass. Automatic Furniture Population of Large Architectural Models. Master's Thesis, Massachusetts Institute of Technology. 2000.

[17] Mathieu Larive, Oliver Le Roux, and Veronique Gaildrat. Using Meta-Heuristics for Constraint-Based 3D Objects Layout. *In Proceedings of the Seventh International Conference on Computer Graphics and Artificial Intelligence.* Limoges, France, May 2004.

[18] G. W. Lecky-Thompson. *Infinite Games Universe: Mathematical Techniques.* Charles River Media. 2001.

[19] J. E. Leigh, A. E. Johnson, T. A. DeFanti. Issues in the design of a flexible distributed architecture for supporting persistence and interoperability in collaborative virtual environments. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, 1–14, 1997.

[20] D. Luebke, and C. Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets, In *Proceedings of the 1995 Symposium on interactive 3D Graphics*, 105–106, 1995.

[21] A. Lindenmayer. Mathematical models for cellular interactions in developement. *Journal of Theoretical Biology*, 18:280-315. 1968

[22] Y. I. H. Parish and P. Muller. Proceedural Modeling of cities.*Proceedings of SIGGRAPH 2001,* 301–308, 2001

[23] Przemyslaw Prusinkiewicz, and Aristid Lindenmayer.*The Algorithmic Beauty of Plants* Springer-Verlag, New York. 1990

[24] Rock Star Games, Grand Theft Auto Series, Website, Jan 2006. http://www.rockstargames.com/

[25] T. Roden, and I. Parberry. Procedural Level Generation. *Game Programming Gems 5*, Charles River Media, 2005.

[26] Alvy Ray Smith. Plants, Fractals and Formal Languages.*Computer Graphics (SIGGRAPH '84 Proceedings)*, 18, 1–10

[27] Sony Corporation. *Sony PSP.* Website. June 2006. http://www.us.playstation.com/PSP/System

[28] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B 7*, 343–361, 1980.

[29] G. Stiny. Spatial relations and grammars. *Environment and Planning B 9*, 313–314, 1982.

[30] Manny Vellon, Kirk Marple, Don Mitchell, and Steven Drucker. The Architecture of a Distributed Virtual Worlds System. *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems.* New Mexico, April, 1998

[31] T. Wang. Integer hash function. Tech. rep., HP Enterprise Java Lab, Website. Jan 2006. http://www.concentric.net/ Ttwang/tech/inthash.htm

[32] Webopedia. Write-back Cache. Website. June 2006. http://www.webopedia.com/TERM/W/write_back_cache.html

[33] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky Instant Architecture, *ACM Transactions on Graphics*, 22, 4, 669–677, 2003.