Procedural Generation of Large Scale Gameworlds

by

Eoghan Carpenter, BSc

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science (Interactive Entertainment Technology)

University of Dublin, Trinity College

September 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Eoghan Carpenter

August 30, 2011

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Eoghan Carpenter

August 30, 2011

Acknowledgments

Many thanks to Dr. Mads Haahr for his supervision and guidance and to Deirdre Glaholm and my classmates for their help and support.

Eoghan Carpenter

University of Dublin, Trinity College
September 2011

Procedural Generation of Large Scale Gameworlds

Eoghan Carpenter
University of Dublin, Trinity College, 2011

Supervisor: Mads Haahr

Gameworlds for modern video games are ever increasing in scale and detail, requiring more and more resources to create this content. The ability to procedurally generate this content allows the creation of gameworlds of such high detail and fidelity in terms of textures, models, characters and other content that they would be impractical to create manually. The procedural generation of this high detail content is already well established in the industry, however less work has been done in the field of generating and managing vast gameworlds.

There are several challenges to creating very large game worlds and this dissertation will focus on two of these. The first problem is being able to produce far more content than could be physically stored in memory but doing so in such a way as to only generate what is needed at a given time in order to minimize cpu and memory utilization. The second problem to be addressed is the need to be able to generate content that conforms to larger structures without the need to generate those larger structures themselves.

This dissertation will present the design, implementation and evaluation of a general

prototype framework that will allow for the procedural generation of arbitrarily sized gameworlds using a system of spatial subdivision whilst allowing for the creation and maintenance of high level structure at low levels of granularity.

To evaluate this framework in action it will be used to procedurally generate galaxies containing on the order of hundreds of billions of stars organized into a commonly found galactic structure. Evaluation will then be carried out on the framework's ability to generate content visually consistent with larger structures at differing levels of granularity. The framework will also be evaluated based on it's performance in terms of memory and cpu usage whilst generating these galaxies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The term Procedural Generation refers to the practice of creating content, often for the production of media, by using algorithms rather than by using manual methods of creating this content. In terms of the game industry this content can cover a number of different products including; 3D models, textures, game levels, sounds and animations. The use of randomness combined with controllable parameters allows the process of procedural generation to produce a large amount of this content, whilst still maintaining a greater or lesser degree of variation in order to avoid repetition.

Procedural generation has been used in video game production for many years. The seminal title, Elite [4], used procedural generation to create 8 'galaxies' each containing 256 planets for the player to explore. In recent years procedural generation is widely used in production and middle-ware products like 3D Studio Max, Adobe Photoshop as well as packages like SpeedTree and Terragen.

1.1 Advantages

As video games continually increase in size and complexity in terms of the volume of content which is expected, as well as the expected fidelity of that content, procedurally generated content is becoming increasingly useful. Many of the environments used in games have become so large as to make manual creation of all content simply too resource intensive. It is impractical for an artist to create a 3D representation of an entire forest and over use of duplication of content can often be quite apparent to

the player, resulting in decreased gameworld fidelity and player immersion. In this case, being able to procedurally generate an arbitrary number of different trees is advantageous.

Similarly, the amount of storage space required to save all this data can quickly become problematic, particularly in the case of console games where such space is limited to what can be fit onto a disc. Procedural generation is in some cases capable of being executed at run-time, meaning that only the algorithms, which are comparatively small, need to be stored but can be used to create more content then could be stored on more media.

Additionally, with the advent of interactive 3D graphics software for internet browsers such as WebGL and the increasing popularity of browser games, the ability to send procedural generation algorithms to the client and locally generate game content rather than transmitting content directly is very advantageous. This can be seen in the browser game Minecraft [5] which uses procedural generation techniques for single player levels.

For the game industry, procedural content generation offers a unique opportunity. That is that the gameworld itself can be tailored by input from, or derived from the actions of, the user. This means that there is the potential there to create a substantively different gameworld for each user and for each play through. This idea is explored by Nitsche et al. [2].

1.2 Disadvantages

Although the benefits of algorithmically generating content are apparent, there are some trade-offs which need to be made.

Even with the use of randomness, procedurally generated content can often produce results with little apparent variety. This can be offset with the use of parameters which enable a degree of artist control over the end results. However, as the scope of variety increases it can be harder to ensure desired results and avoid unpredictable behaviour.

For this reason, the algorithms themselves which are used for procedural generation can often be very difficult to create. Trying to abstract the generation of content into a series of procedures is often an unintuitive process and when increasing the numbers of artist tunable parameters, ensuring stable and believable results can be a difficult

task.

Furthermore, savings in terms of storage space is offset by the need for more processing required for these algorithms to run. In computer games, processing time is as much at a premium as storage, so increasing usage for procedural generation will reduce the cpu budget for other tasks like physics simulation and Artificial Intelligence.

1.3 Types of Procedural Generation

The field of procedural generation is a broad one. There exists many methods and techniques, and some of these are better suited than others to particular content types. The following is a brief description of some of the more widely used procedural techniques:

1.3.1 L-Systems

Simply put, L-systems (Lindenmeyer Systems) are a string of characters from some user defined alphabet and a set of production rules. A production rule is a function which will modify a string A into a new string B (A and B being ordered tuples) and the L-System iteratively makes this modification to the string based on the production rules.

This method is commonly used for the procedural generation of plants. Lluch et al. [6] provide an example of this in their work on multi-resolution plants and trees, in this case using 'axioms' and 'productions'. When a production is applied to the axiom, a new derived chain results. For example, an axiom contains the details required for the representation of a branch on a plant. One production will generate two children (new branches) and another will halve the length of the branch for these children. Further more the production can be bounded by constraints after which the iteration will stop, in this case suitable bounds may be a minimum branch length or a maximum number of children.

1.3.2 Grammars

A grammar is a set of formation rules in a formal language used to generate strings. These rules generate strings in accordance with the grammar's syntax, but the resulting

string has no inherent meaning. In this way, L-Systems can be described as a particular kind of grammar. In Instant Architecture, Wonka et al. [7] use a combination of a shape grammar (a grammar in which rules are applied to shapes rather than to strings) and a split grammar (one which includes operations to split these shapes into meaningful chunks). By using a large database of grammar rules they were able to model a variety of designs from this single rule set rather than attempting to construct a grammar for each model.

By iteratively splitting a shape (such as a building's facade) into smaller shapes according to rules which would allow for the creation of window and door shapes (for example) they were able to construct a huge variety of differing building designs.

The system could also be easily extended as Larive et al. have shown [8]. In this paper they applied a similar split technique to 'walls' rather than shapes allowing them to choose production rules based on the type of wall being split (Abstract walls, Wall Panels, Bordered Walls, Extruded Walls etc). This enabled them to achieve an even greater degree of control over, and variety in, the buildings being generated.

1.3.3 Fractals

Ebert [9] explains techniques for using fractals and multi-fractals to generate content like clouds, mountains and others. He describes a fractal as being "a geometrically complex object, the complexity of which arises through the repetition of form over a range of scales" [9, p. 431]. In particular Ebert uses a fractional Brownian motion (fBm) type of fractal to generate content. These can be used to generate a type of noise (similar to Perlin noise) which is fractal in nature, i.e. they will generate a required granularity of detail over any scale by adjusting the spatial frequency of peaks within the noise generated and the amplitude of the noise. The fBm is characterized by its 'power spectrum' which charts how the frequency and amplitude are related.

1.4 Current Challenges

As has been stated, there has already been much work done in the field of procedural content generation. However, the majority of this work has been focused on high detail, high fidelity content production and tends to be very result specific. Much of this work

is also proprietary and not available for public use. This means that it is often the case that the organization of processes is non-standard and the code is often not reusable. Little work has been done at addressing these issues of standardization and reusability, which with the increasing use of procedural content generation in games, will become increasingly problematic.

The two main challenges which will be addressed in this dissertation are:

- Generating 'Structure' at varying levels of detail which will be consistent with the overall structure of the generated content, and doing this at run-time. The term 'structure' in this sense is used to denote the data or visuals produced by the procedural generation at a particular level of detail which must conform to that produced at other levels of detail. For example, an algorithm for generating build interiors must be able to produce a room which will 'fit' with other structures within the building, doors must line up with hallways, the room cannot overlap with stair-wells etc. Some methods of procedural generation do this inherently, for example fractal based generation, but this remains a difficulty for other methods.

- The second challenge for this dissertation will be minimizing the overhead used in terms of memory and processor utilization. Although procedural generation can grant huge benefits, it must still be capable of running within an environment where these resources are strictly budgeted.

1.5 Goals

This dissertation will attempt to create a software architecture capable of addressing the challenges outlined in section 1.4. In order to evaluate the success of this architecture, it will be used to create a procedural galaxy generator and evaluated by the following criteria:

1. The architecture should be capable of imposing structure, as defined in the previous section, upon any resulting procedurally generated gameworld.

2. The architecture must be capable of generating the required content whilst itself remaining as light weight as possible in terms of memory usage and processing

time. By doing this it should be capable of being used within a resource tight application, such as a computer game, to create and manage large scale gameworlds without unduly increasing resource usage.

3. The architecture should provide a template for the organization of procedural generation algorithms which is flexible enough to be used in a variety of situations, which is extensible and capable of being used with other systems.

1.6 Document Roadmap

The remainder of this dissertation will be divided in the following sections:

- Chapter 2 will review a selection of the literature on the topic of procedural generation, specifically how it pertains to the goals of this dissertation.
- Chapter 3 will analyse the methods and techniques used in this literature in order to design and present a general architecture for procedural generation and the framework which implements it.
- Chapter 4 will examine the creation of the procedural galaxy generator developed using the proposed architecture.
- Chapter 5 will evaluate the framework and the resulting prototype galaxy generator and assess their success at achieving the stated goals.
- Finally, Chapter 6 concludes the project, puts its results in context and identifies areas of future work.

Chapter 2

State Of the Art

Procedural content has been around for some time. Marshall et al. [10] developed a system for procedural generating large terrains in 1980. The original Elite [4] computer game procedurally generated 'galaxies' consisting of 256 stars in 1984. As such the field is already well established, in particular for generating things like terrain, textures, sounds, buildings and foliage. However, the specific challenges of this dissertation relate more to the control and organization of generated data rather than its generation. This is something which has not been nearly as well explored and as such there is a limited amount of existing literature. As a result, the following sections will examine the techniques used in a selection of papers to address the challenges outlined in section 1.4. In Chapter 3 these techniques will be examined in order to design the general architecture for procedural content generation.

2.1 Dynamic Landscape Generation using Page Management

Danaher [11] in 2002 proposed a novel system for maintaining large amounts of generated terrain using a page management method which he refers to as 'the spherical offset method'. To maintain a terrain around the user, the system divides the terrain into a series of blocks referred to as pages each of which contains procedurally generated terrain based on a height-map, and which can then be displayed as required. Thus as the user moves about the landscape new pages are created and added as required

rather than reproducing the entire scene around the user.

The management of these pages is based on a spherical page wrapping approach. The overall terrain is composed of 'submaps' and as the user approaches the border of one submap a tessellated (to prevent discontinuity) neighbour can be generated and added to the map, with submaps no longer visible being removed. By using this method far more efficiency could be obtained than with older methods, as the 'map' was no longer centred on the user and so only needed to be updated as the user approached a border rather then on a continuous basis. Furthermore as an edge was approached three new submaps could be generated and replace three existing submaps within the map, ensuring a limitation on memory usage.

2.2 Real-time generation of 'pseudo-infinite' cities

Greuter et al. [12] in 2003 proposed a system to generate what they describe as 'pseudo-infinite' cities. This was capable of procedurally generating buildings (including basic interiors) on the fly and as needed by the user. The shape of these buildings is determined by their location, ensuring that if the user returns to the same area the same buildings will be generated.

The system uses a method of view frustum filling similar to that used to clip scenes in a typical 3D application. In this case the view is filled with procedurally created geometry rather than pre-built models but this allows the deletion of geometry which is not visible, allowing the application to maintain roughly constant memory usage. The frustum is filled with 2D square tiles to determine visibility and generates buildings based on these tiles. Although this allows for very quick calculation and generation, the drawback with this method is that the layout of the city will remain in an unrealistic grid pattern.

Once these tiles are calculated, the positions of the tiles are passed through a hashing function based on Wang's [13] 32bit Integer Hashing Function, where they are hashed with a global 'citySeed', in order to generate a seed to be used for the pseudo-random number generator whilst generating that particular tile. This allows adjacent tiles to be significantly different from each other whereas without this hashing step, similar locations would result in recognisably similar content being generated.

Of particular interest is the methods used by Greuter to minimize usage of and

maximize efficiency of memory for this application. As the system is inherently a 'lazy evaluation' system, only areas currently visible to the user need to be maintained. In order to achieve this, a least recently used (LRU) caching policy was adopted. This cache performs three main tasks. Determining if a particular item is in the cache, determining the least recently used item and inserting new items. To implement this a combination of the C++ standard library's doubly linked list (std::list) and balanced tree (std::map) structures were combined. The list is ordered by access time and the map provides a fast index into the list. This indexing provides an increase in performance of queries from O(n) to O(log n). The overall container itself is organized by key value pairs. The key is the integer identifier for a particular building and the value is the data for that particular building (including the OpenGL display list for the building, hence avoiding the need to re-generate the geometry at each update). The map can then be used to look up entries in the list based on this key. By using this system it becomes quite easy to control how long items are stored by specifying a maximum time to live for entries and to control memory usage by specifying the maximum capacity of the LRU structure.

2.3 Persistent Realtime Building Interior Generation

Hahn et al. [1] extended the ideas of Greuter above in order to generate building interiors during runtime which allow a seamless walk-through of the building using only a fraction of the memory which would be required for the entire building, and thus provides an interactive environment that is much larger than the available memory and much larger than a developer would be capable of creating manually in a reasonable amount of time.

In order to achieve this, building interiors are divided into two types of region:

- Temporary Regions, which are regions of space where generation can occur, represented by axis aligned bounding boxes.
- Built Regions, which are the final visible product of the generation process. These hold the geometry needed for rendering and collision detection as well as any

visible objects placed within the building.

In this case, lazy generation of the building interior can be exploited to a high degree due to the natural occlusion of such a scene. Under normal circumstances this would cause the order that the regions are generated to be dependent on the path taken through them, resulting in different buildings being created by taking different paths. In order to avoid this problem, the generated interiors must be independent of this order. To achieve this, a top-down approach is used allowing temporary regions to be created independently of the path of the user and any other regions. Because of this, regions can be generated in any order and will not affect the final outcome.

Variation is achieved using a pseudo-random number generator (PRNG) with an independent seed for each region. To calculate the seed for a given region, the midpoint of the region and a global seed are used in a similar fashion to that used by Greuter [12]. The three coordinates of the midpoint are rounded to the nearest integer and then hashed with the global seed to create a new seed for that region.

The top-down hierarchical approach used in this work is as follows:

- Building Setup. The initial seed for the building is used to generate features which will affect multiple floors of the building (stairwells, elevators etc) as well as features which will affect global aspects of the building like the textures to be used.

- Floor Division. This stage divides the building into uniformly distributed floors, using a binary subdivision method.

- Hallway Division. These hallways are constructed by dividing regions around other rectangular regions into either straight or looping hallway regions. This will continue until a region reaches the minimum size allowed for a room cluster.

- Room Cluster Division. The regions found between hallways are then divided into rooms. This is dependent on 'portals' coming into the region, and will continue until a valid divider cannot be found or until there is one portal for each room.

- Built Region Generation. This is the final stage of the generation process in which a visible region is created. The geometry is created based on the bounding box boundary and the portals are attached to this.

As can be seen, a similar top-down approach may be capable of addressing the problem of achieving structure throughout a procedurally generated system outline in section 1.4.

Memory is managed in Hahn's by using a generation tree, providing a history so that the generation process can quickly be reversed. Each node of the tree represents a past step in generating the building interior and contains an axis aligned bounding box, a region type and connections to its parent and one or more children. Each leaf on the tree also represents a region of the building which allows quick access into the needed parts of the tree when regions are deleted and efficient point locating. This point locating allows the system to quickly find which region a given point lies within (by the axis aligned bounding boxes) in logarithmic time.

Memory usage is limited by deleting built regions when they are no longer needed. When a region is deleted, the system moves recursively up the tree until the largest subtree is found which does not contain any built regions and which can be replaced by a single temporary region. Further, the caching system is based on a Least Recently Used system similar to that described by Greuter.

Finally the paper also describes a system for allowing persistent user changes to the generated content, storing changes made in a hash map and allowing them to be retrieved when a given region needs to be regenerated.

2.4 Designing Procedural Game Spaces: A Case Study

Nitsche et al. [2] present a game prototype, called Charbitat, which attempts to integrate procedural space generation with gameplay, resulting in a game in which the player creates the gameworld as they play through it. In order to provide a scalable and consistent gameworld the environment is split into large tiles, each of which contains its own seed value for procedural generation of terrain. This terrain is then mixed with global landscape features (rivers, lakes etc) in order to ensure continuity and populated with premade assets (items, creatures etc). The seeds used for these tiles are dependent on player specific choices. The actions of the player influence the seed values and hence influence the generation of the gameworld. For this reason the gameworld cannot be

generated in its entirety from an initial seed as is the case in the papers described above, and a given world must therefore have all the seeds and the positions of the tiles they refer to stored individually. For this reason, the memory usage goes from O(log n) as in the case of Hahn's [1] buildings which are generated from a single seed, to O(n). Clearly, this method for gameworld generation is not well suited for large gameworlds but does present an interesting approach to the creation of a gameworld that an ideal software architecture should be able to cope with. Although this dissertation is not focused on maintaining persistence in procedurally generated gameworlds, this style of gameworld generation and the problems of creating a software architecture which supports it will be discussed in chapter 3.

Chapter 3

Architecture Design and Implementation

As can be seen in Chapter 2, there are many ways of procedurally generating content that may be used in a computer game. This presents the problem of designing an architecture which will be able to cope with different methods of procedural generation in an organized way. In Section 3.1 some of these issues will be examined in an effort to create an architecture which is flexible and general enough to incorporate these different methods. In Section 3.2 an architecture will be described which it is proposed will be capable of meeting the challenges described in section 1.4

3.1 Design Issues

3.1.1 Determinism

Procedural content generators are capable of generating both deterministic and non-deterministic content. That is to say that they can generate content which will be identical each time it is created or which will be different each time it is created.

This issue is addressed by Hahn et al [1] when discussing the generation of building interiors. In this case it was required that the building layout be constant despite the path that a user would use to walk through the building, hence despite the order of generation. In order to address this issue the authors implemented a top-down system which would spatially divide the building into separate, independent regions any of

which could be generated independently of any others. This necessitated the creation of a minimal amount of data for the entire building beyond what was immediately required for the currently generated region but this overhead was minimal enough to be practically maintained while ensuring order independent generation.

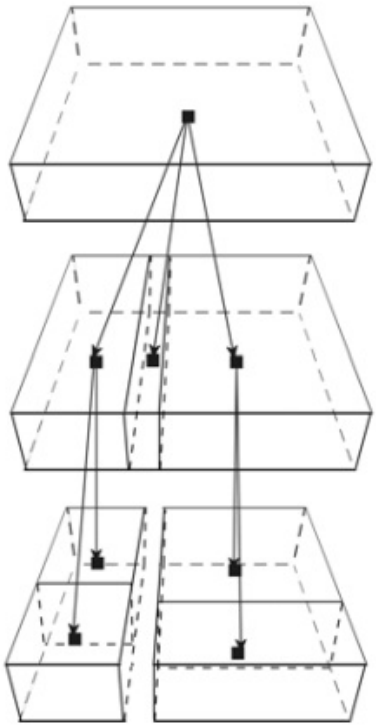


Figure 3.1: The top-down spatial subdivision of a building. Hahn et al. 2006 [1].

By comparison, Nitsche et al. [2] implemented a game world which would be unique to the path taken by the user when exploring that world and so creating a differing world at each play through. To achieve this, they implemented a tile system for the game world such that the world would be created in discrete 500 by 500 metre areas. Each of these tiles would be assigned a seed as the player entered them and would be generated on the fly as needed, and where the seed would determine the layout and content of that tile. The seeds would be determined by the actions of the player, in this case the player choosing a path based on Taoist elements, and the generation of new tiles would reflect the choices of the player. Hence the system would eventually achieve a list of seeds and locations for the tiles but which would be dependent on the

path chosen by the player and so different at each play through.

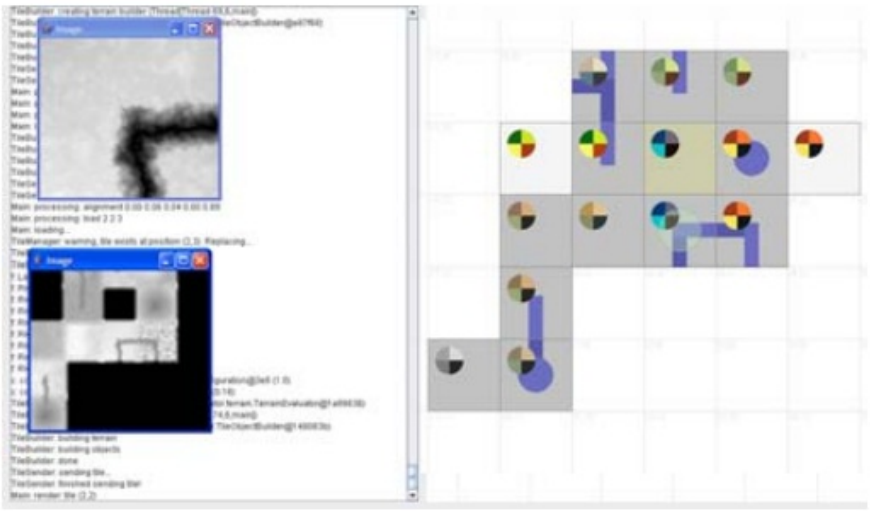


Figure 3.2: Path independent tiles. Nitsche et al. 2006 [2].

The architecture should be able to account for both the circumstances described above. The more limiting factor here is the need for order-independent generation as with Hahns buildings so its seems logical that the architecture is built around a similar top-down tree structure.

Although this can be done straightforwardly in a situation like that described by Hahn, such a design is a little bit less intuitive when applied to that of Nitsche's path-dependent game world. In this case, generation works from the root out according to the path the player follows. This can still be accounted for using the tree pattern in a slightly more abstract fashion. An example of how this might work can be seen in Figure 3.3, where the tree is still based on a spatial subdivision technique but this time is grown in line with the players exploration of the world. This can result in a lot of nodes on the tree covering empty space, however these nodes do not need to have any content generated for them so the overhead will not need to be too costly. This does point to the idea that any kind of general architecture must be capable of growing on the fly and having differing depths at different sections of the tree.

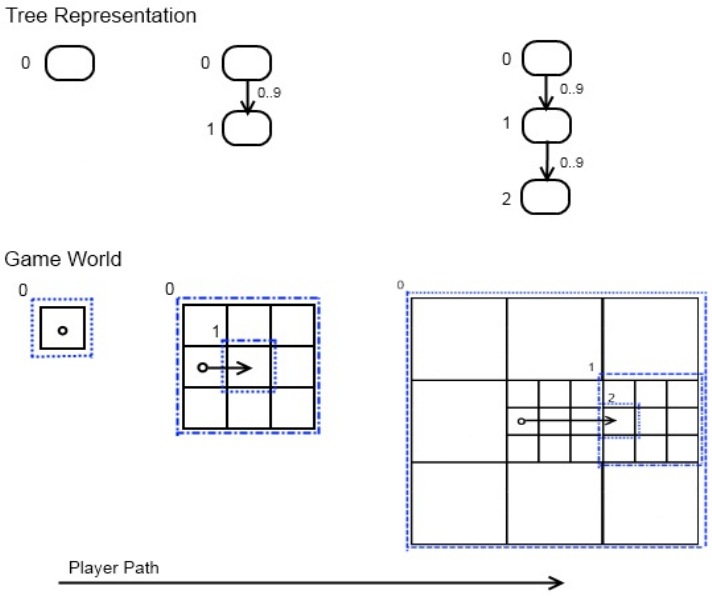


Figure 3.3: Using a top down architecture to represent a path-dependent game world. As the player moves through the world, the depth of the tree grows dynamically.

3.1.2 Lazy Evaluation

As stated in Chapter 1.5 this dissertation is concerned with the creation of large scale gameworlds. As such, it is quite possible that more content could be generated than could ever be held in memory at once. For this reason, the architecture should be capable of generating only what is needed at a given time. This strategy of limiting what is generated is referred to more generally as 'lazy evaluation'.

Greuter et al. [12] were able to achieve this for their 'pseudo-infinite' cities by limiting generation to a frustum representing the users point of view. The procedural generation of buildings in these cities was based up the location on a grid of those buildings and only buildings whose location was with the frustum were generated, and were generated from the apex outward. As buildings were generated they were added to a Least Recently Used (LRU) list which would handle memory management by removing buildings no longer required. The combination of these two techniques would mean that the memory usage of the system could be readily capped by limiting the size of the LRU list and would ensure that only content which was visible would ever be generated.

Hahn et al. [1] used a different strategy for memory management which was integrated into their tree pattern procedural generation. In this case, the architecture lends itself naturally to lazy evaluation. By dividing up the building spatially down to the point where nodes on the tree represent individual rooms, the authors were able to easily determine which rooms were required for visualizing the scene at any time and only generate these rooms. Since this system was also purely deterministic (order of generation independent), any rooms which had been previously generated but which were no longer required could easily be discarded and regenerated at a later point.

3.1.3 Structure

The idea of structure in this sense is a slightly abstract one. It refers to the need to generate procedural content for a given point in the game-world which will 'fit' with the rest of the game world. For example, if generating a room in a building, it should be ensured that doorways will be shared with other rooms or corridors, windows should be shared with outside walls and the room should not overlap other rooms. The difficulty in this is generating this data without generating unneeded data from surrounding portions of the game world, ensuring a room will not overlap with other rooms without needing to generate those rooms.

Again, Hahn et al. [1] were able to address this with the tree pattern that was used for generating the buildings. The spatial subdivision strategy began with a single bounding volume which covered the entire building. This was then subdivided through a series of discrete steps which would assign both space and purpose to the child nodes being generated. First, building wide spaces like elevator shafts and stair wells were assigned. Next the space would be split into individual floors and within these floor wide spaces like hallways would be assigned. This process would continue down until individual spaces represented individual rooms, each of which would fit well with the surrounding rooms.

Although this solution provided the ability to represent larger structures at the point of generation, the process taken of dividing and assigning space is a rigid one specific to this particular context. As such it would not be reusable in another game-world and is not by itself flexible enough to be re-tasked. In order to be capable of achieving the same results but in a flexible enough way to be applied to a variety of

situations, this principle of assigning purpose at different levels of the tree will need to be generalized such that subdivision/assignment rules can be implemented at any point of the generation algorithm.

3.1.4 Scalability

The subject of this dissertation is the procedural generation of large scale game worlds. As such, the ability of the resulting architecture to scale to an ideally, arbitrary size, and to be efficient and practical in doing so, is a crucial one.

Greuter et al. [12] were able to generate pseudo-infinite cities which would meet the requirement of being scalable and as the buildings generated were dependent purely on their position this process was also efficient. However, this system did not account for any kind of structure to the city above individual buildings laid out in a grid pattern, i.e. each building depended solely on its position rather than any surrounding features.

Nitsche et al. [2] used an architecture which simply used a list to store generated tiles for their game-world. Furthermore, as these tiles were dependent on the user's path through the world they could not be reliably regenerated if they were discarded. At the least some situational data needed to be kept for every tile generated. This means that the system would grow linearly as the player progresses and as such scales poorly.

Hahn et al. [1] used a tree structure for their building generation. Tree structures generally offer good scalability, however this ability to scale was in this case limited by the steps of assigning purpose to space. A building could be made to arbitrary size, but would still follow the same pattern of building to floor to room, just with more nodes at each level of the tree.

From examining these cases, it seems apparent that a good system for achieving scalability is the tree structure. This is a data structure which handles scaling efficiently and practically. However, to avoid the rigidity of Hahn's approach, the tree should be generalized in order to deal with a variety of situations from spatial subdivision to more abstract situations. In other words the type of tree used in a given situation should be driven by the specifics of that situation rather than by the design of the architecture. For example, specifying a Binary Tree in the architecture would make little sense in the case shown in Fig 3.3 where a 9-Tree would be more applicable.

For this reason, the termination of growth of such a tree must also be addressed in any architecture. It may be desirable to have different circumstances under which the tree will be terminated and content generated, at different points in the same tree. As such, the the architecture should be able to support this.

3.1.5 Randomness

As stated in Chapter 1 one of the greatest advantages of procedurally generated content is its ability to generate a degree of variety in its results. A procedural tree generator would be of limited use if it only produced a single tree repeatedly. For this reason, randomness is essential to procedural generation, the parameters of the generated content must be in some part dependent on a random value to achieve variety. However, as discussed in Section 3.1.1 in many cases determinism is also a prerequisite, i.e. the system should be able to produce variety, but the same content must be reproducible if necessary.

To achieve this a pseudorandom number generator (PRNG) should be used by the architecture. These generators will always produce the same sequence of random numbers for a given seed. By using such a system, as long as the seeds are preserved or determined in a reliable fashion and the steps taken in producing the content are carried out in the same order, the resulting content should be identical while still supporting the requirement for variety.

This can be seen in the pseudo-infinite cities of Greuter et al. [12] where the seeds for each building were based on the buildings location and so the same building would always be reproduced at a given location, but a different building could be generated at each location.

It is therefore an important aspect of the architecture used in this dissertation to allow for tight control of both the seeds and the pseudorandom number generation process to ensure both variety and determinism in any implementation.

3.2 Proposed Architecture

Based on the requirements and analysis from the previous section, an architecture was designed to address all these issues. Fig. 3.4 shows a overview of the main parts of

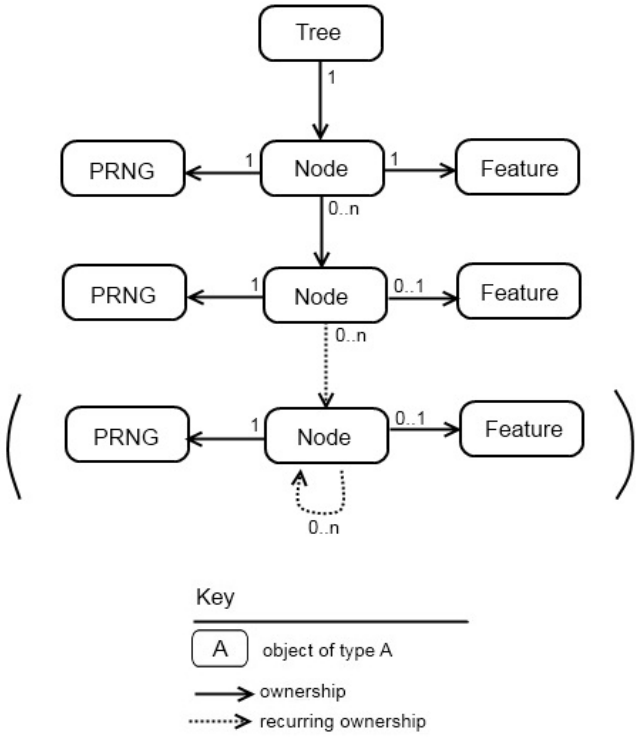


Figure 3.4: Diagram of the proposed architecture.

this architecture, which is based around a tree structure. Generally speaking, the tree data structure provides a scalable and efficient pattern for data representation and management. Tree structures are widely used throughout computer science subjects and common examples include the Octree and the Binary Spatial Partitioning tree. In most cases the particular type of tree used is based on the implementation specific requirements. In this case, the architecture is designed to be as general and flexible as possible so instead an N-Tree structure was chosen. This is a tree which does not have a fixed number of children for each node, nor a fixed depth. This introduces some extra complexity during programming as the number of children or the depth can vary across the tree so care must be taken to ensure valid operations.

Each node in the tree contains its own pseudorandom number generator (PRNG). Although this creates slightly more overhead in terms of memory usage compared to using a single PRNG for the whole tree, it does make it more straightforward to ensure that random numbers can be generated in a controlled and predictable way, i.e. there

is no chance of one node changing the order of number generation of another node and the pseudorandom number generators do not need to be constantly reseeded. It also allows for the easy changing of random number generation methods without the need to alter any other code and allows different random number generation methods to be used at different points in the tree if there were such a requirement.

Each node may also contain up to one feature. The feature is the work horse of this architecture and will contain the logic for procedural generation, handle the placing of structure in the tree and contain the code for node subdivision and subdivision termination in the subtree below it. This means that features can appear in any part of the tree (although the root node must contain a feature) and these will impose the structure referred to in section 1.4 upon all nodes below it, unless they too contain a feature. In other words, when a call is made to a node to carry out procedural generation on a particular node, if that node does not contain a feature itself, a search is carried out up the tree until a node with a feature is found and the generation code of that feature is used. This is depicted in Fig. 3.5. In this case a hypothetical 'Generate()' function is called for the node at the bottom of the tree. Since this node does not contain a feature of its own, a search is done back up the tree until a feature is found to carry out the procedural generation, in this case Feature2.

This approach allows the imposition of structure over the tree in a similar manner to that used by Hahn et al. [1] but to do so in a much more flexible way. It should be noted that any given node in the tree may be associated with only a single feature which is a limitation of this architecture.

3.3 Creating the Framework

With the architecture designed as in section 3.2 it is now possible to implement this architecture into a framework. This was done using the C++ programming language, although any similar object oriented language could be used such as Java or C#. To achieve this framework, interface classes were created for each of the elements of the architecture which specify the minimum required parameters and functionality for the architecture to operate. These interfaces are INode, ITree, IFeature and IPRNG and the following sections will describe them in detail.

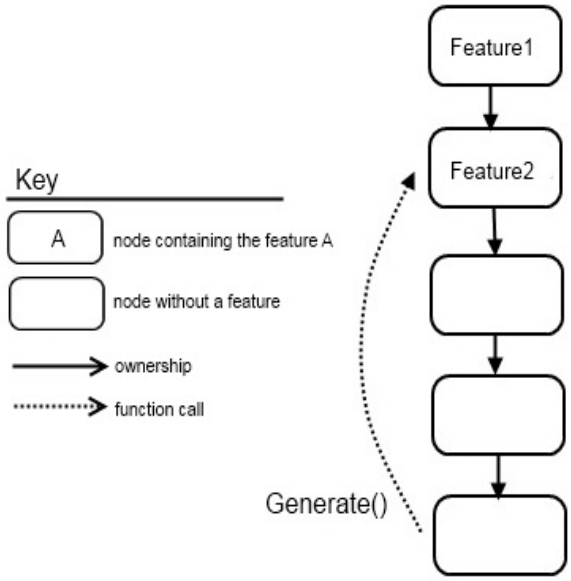


Figure 3.5: Using the feature for procedural generation calls.

3.3.1 INode

The tree nodes used in any application of this framework can be customized by the user but must implement the INode interface to integrate them with the rest of the framework. The following is a list of properties of the INode interface and the virtual functions which must be overridden.

INode* m_parent this is a handle to this node's parent node (this will be null for the root node) and this is stored to enable the application to walk up the tree.

vector⟨INode*⟩ m_children this is a vector containing pointers to any child nodes associated with this node. The vector reflects that this is an N-Tree and may have a variable number of children.

IFeature* m_feature each node may have a handle to up to one IFeature implementation. The IFeature interface is explained in section 3.3.3.

int m_depth this is the depth of the node within the tree, i.e. the number of steps that must be taken down the tree from the root node in order to reach this one. This value is used in conjunction with the IFeature.

IPRNG* m_prng this is a handle to the Pseudo Random Number Generator for this node. The IPRNG interface is explained in section 3.3.4.

IFeature* GetFeature() this is a function which must be overridden to return the IFeature that is to be applied to this node. If this node does not contain a feature it should search up the tree until an IFeature is found and return this. By design there must always be at least one IFeature implemented in the tree at the root node so that this function may always be able to return a value.

3.3.2 ITree

The ITree interface is the basic control structure for the tree architecture. The following is a list of properties and virtual functions associated with it.

INode* m_currentNode this is a handle to the node which is currently active, i.e. being split, generated, visualized etc.

ITree(long seed) this is the constructor for the tree structure which accepts an initial seed used for the root node.

void NextLevel(int child) this function sets the m_currentNode to point at the selected child of the current node.

void PreviousLevel() this function sets the m_currentNode to point at the parent of the current node.

void Prune() this function prunes the tree at the currently selected node removing all unnecessary data.

void Generate() this function calls the procedural generation algorithms on the currently selected node.

3.3.3 IFeature

The IFeature interface is the implementation of the feature element of the architecture. An implementation of this interface will contain the logic for procedural generation, handle the placing of structure in the tree and contain the code for subdivision and

subdivision termination. The following is a list of properties of the IFeature interface and the virtual functions which must be overridden.

int m_ownerDepth this is a record of the depth down the tree that the INode this IFeature is associated with is placed. This is used to calculate the relative depth of a particular node in the subtree which is used for the generation of procedural content and structure.

void GenerateChildren(INode* node) this function generates children for the given node. This would include assigning a seed value and any other implementation specific parameters for the children such as dimension information for a spatial subdivision scheme. It is important to note that this does not generate any content for the system, but only handles the splitting of the tree.

void GenerateChildren(INode* node, int child) this is the same as the above but handles the generation of a single particular child.

void GenerateLeaves(INode* node) this function contains the code for procedural generation of the final content. For example, in a building interior generator, the leaf nodes may each account for one room. This function would then handle the generation of these rooms.

void Generate(INode* node) this function is slightly more abstract in nature, but handles the generation of data which is not the final data. An example of this in use will be seen in Chapter 4 where the distribution of stars in a galactic feature is estimated in order to give the impression of structure at different levels of detail without generating the individual stars themselves. In the case of a building generator, this may give a simplified, estimated layout of the room or rooms contained in the given node.

void Terminate(INode* node) this function contains the logic for terminating the subdivision of nodes and generating the final content. This logic is again implementation specific but an example would be terminating splitting when a single node contains a single room in a building interior generator.

The idea of placing all the procedural generation code into the IFeature class may seem to be counter intuitive but doing so grants several advantages.

- It provides a single place and format for all procedural content logic rather than having unformalized logic distributed throughout the system.
- It allows for structure to be placed at any point throughout the tree and at any scale. In the case of a world generator, these structures could take the form of continents, oceans, deserts, cities, forests etc. As the size and position of these structures will be variable, placing them at any point in the tree allows the programmer to impose structure similar to that used for splitting space into floors, hallways and rooms used by Hahn et al. [1] but to do so in a much more flexible way.
- In many cases, the final content generation will be dependent on the nature of the structure that it is a part of. The content generated at the leaves of a desert structure will be significantly different to those of a city structure. By placing this content generation logic here it associates this structure and the resulting procedural content generation logic together in a single location.

3.3.4 IPRNG

Each node contains its own instance of an IPRNG (Interface Pseudo Random Number Generator) implementation. The IPRNG interface has the following properties and virtual functions.

long m_seed the seed of the node the IPRNG is associated with. Although this is duplicated data from the node, it was felt that having a copy here would mean that the IPRNG would not need to be continuously referring back to its owner node to access this value.

void Reset() this function resets the IPRNG implementation back to its initial state, in effect reseeding itself. This allows for the regeneration of data for a particular node deterministically.

long GetNext() this returns the next pseudo random number produced by the IPRNG. In practice this function should use a template data type so as to be configurable to the specific implementation.

Chapter 4

Galaxy Generator Design and Implementation

In order to demonstrate the architecture and framework described in the previous chapter and to enable their evaluation against the goals stated in section 1.4, it is necessary to create an application which utilizes this framework for the generation of large scale gameworlds. It was decided to implement a procedural generator for a realisticly sized galaxy to do this. Such a generator should be capable of creating a galaxy containing on the order of hundreds of billions of stars in order to demonstrate the scalability, efficiency and consistency of generation with this framework. The rest of this chapter will describe the creation of this galaxy generator. Section 4.1 will cover the parameters and design choices used for the creation of a galaxy. Section 4.2 will detail the specifics of how the application fits the described framework and how the eventual stars and planets are generated.

4.1 Procedural Generation of Galaxies

Galaxies exist in many forms, regular, irregular, spiral, barred spiral, dwarf etc. One of the most recognisable of these galaxy types is the spiral galaxy such as that shown in Fig. 4.1 and of which our own Milky Way is an example. For this reason it was chosen to use the described framework to procedurally generate spiral arm galaxies of this type and of a similar scale. The Milky Way is estimated to contain somewhere

between two hundred and four hundred billion stars. For this application it was chosen to use the figure of three hundred billion stars. This many stars should show the ready scalability of the framework though it should be noted that the framework is intended to be capable of generating much larger gameworlds than this.

Figure 4.1: Pinwheel Galaxy, Europe Space Agency & NASA.

In order to represent structure in this galaxy, three implementations of the IFeature interface will be used. These are described in detail in the following subsections.

4.1.1 GalacticFeature

This IFeature represents the structure of the galaxy as a whole and will always be associated with the root node of the tree. The Generate() function of this implementation will contain code for the visual estimation of the shape of the galaxy as a whole. This visual estimation will consist of a central bulge of stars at the core of the galaxy and four spiral arms radiating out from this core. As well as this, the feature will contain the necessary logic for dividing the space up and assigning it to its children. In this case the feature is limited to creating galaxies of the same four armed structure. Although using different seeds will result in different content being generated this structure will remain the same. This is not a limitation of the framework or architecture and there is no reason why an application with randomized structure could not be used.

4.1.2 SpiralArmFeature

Spiral arms are regions of stars which extend from the centre of spiral galaxies and which resemble spirals. These arms tend to contain younger hotter stars, as well as smaller galactic features such as open clusters. To represent this spiral arm structure the SpiralArmFeature will use a simple spline for the curve of the arm. This curve can then be used for calculating the distribution of stars at this point in the tree and for nodes in the subtree.

4.1.3 ClusterFeature

The ClusterFeature will be used to represent the Galactic Bulge at the centre of the galaxy. This bulge is a large, tightly packed cluster of older, cooler stars found in the centre of most spiral galaxies. To represent this structure the ClusterFeature will use a radial distribution system for the distribution of stars at this point in the tree and for nodes in the subtree.

4.2 Creating the Application

4.2.1 Subdivision

The most obvious way of subdividing this kind of tree is to do so using spatial subdivision techniques. Octrees are a common tree data structure used for recursively dividing three dimensional space into eight octants. This can be seen in Fig. 4.2. Although this method does allow for the efficient splitting of space, it does not account for structures in that space. In other words, to use this method directly would result in structures like spiral arms being divided across multiple sections rather than wholly in a single section. This would make the process of estimating shape and the distribution of stars in these sections considerably more difficult. For this reason the GalacticFeature subdivides itself into five children, each of which wholly contains another structure, one ClusterFeature and four SpiralArm features. This splitting method is depicted in Fig. 4.3. Doing this makes the process of representing shape in the subtrees of these features considerably more straightforward. The ClusterFeature and the SpiralArmFeature themselves divide on the octree pattern although if there were other features

situated below these on the tree they could use their own subdivision methods. One of the advantages of using this architecture is this ability to implement different subdivision techniques which are appropriate to the structure at different points in the tree.

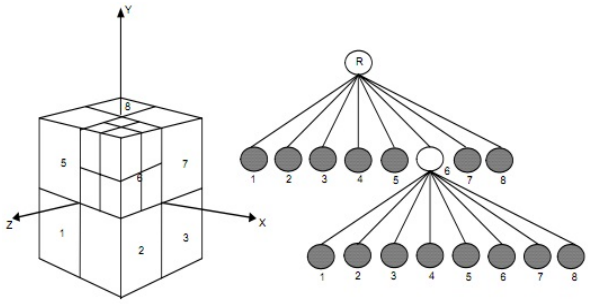


Figure 4.2: Splitting of three dimensional space in an octree. Zhang & Xu, 2006 [3]

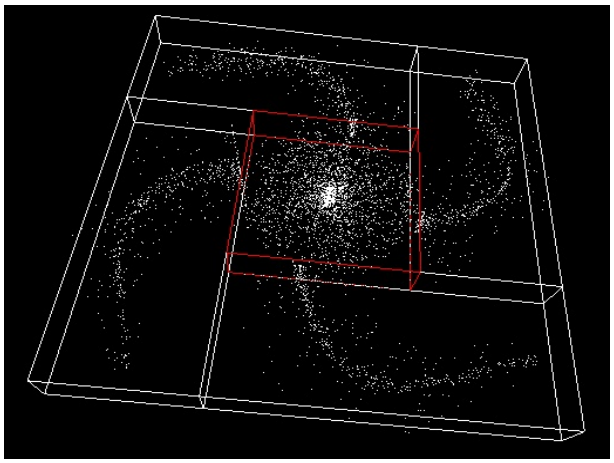


Figure 4.3: A galaxy being spatially divided into 5 children.

4.2.2 Tree and Node Implementations

This demonstration project uses one implementation of the ITree interface, MyTree, and two implementations of the INode interface, MyNode and SSNode (Solar System Node). The following sections will describe how these implementations work and how they extend the interfaces for this project.

MyTree

The MyTree class implements the virtual functions described in section 3.3.2. Additionally it contains the following two project specific functions.

MyTree(std::string address) This is an additional constructor for the tree structure which will enable it to generate a specific galaxy to a specific level of detail using an address in the form of a string. An explanation of the address system can be found in section 4.2.4.

std::string GetDisplay() This function gathers information about the current node and tree state including the number of stars in this node, the number of nodes active in the memory amongst other things in the form of a string for display purposes.

MyNode

The MyNode implementation of the INode interface is the node class used throughout the tree with the exception of the final leaves. As well as implementing the functionality of the INode as described in section 3.3.1 the following properties and functions have been added to support 3D spatial partitioning and other projects specific requirements.

MyNode(long seed, std::string address) This additional constructor allows for the passing of both seed and address values during the creation of nodes for the tree.

void Draw() This is an additional function used for visualization of the galaxy. More details can be found in section 4.2.6.

double m_numStars This tracks the numbers of stars contained in the volume represented by this node.

Vector3D m_centre The 3D spatial coordinate for the centre of the nodes volume.

double m_height, m_width, m_length The dimensions of the space bounded by this volume.

SSNode

The SSNode is used for the leaves on the tree and contains the finally generated data for this application. It should be noted that the choice to terminate the division of the tree at this point was purely a design choice. In another application, there is no reason why subdivison could not carry on down to individual stars, planet or beyond using a solar system or planet feature to alter subdivision at this point. As well as implementing the virtual functions of the INode interface the SSNode also contains a vector of pointers to Star objects. Each SSNode can have between one and three stars to represent unary, binary and trinary solar systems.

4.2.3 Pseudorandom Number Generation

The implementation of the IPRNG interface, MyPRNG, uses a Linear Congruential Generator originally written by Gardner [14]. A linear congruential generator is a well established method for generating pseudorandom numbers. In it can be represented by the equation:

$$X_{n+1} = (aX_n + c)(mod\ m)$$

where X_n represents a value in the sequence of pseudo random numbers (X_0 being the seed value), a is a multiplier constant, c is an additive constant and m is the modulus. Gardners algorithm provides a fast an efficient method of generating pseudorandom long values with a period of 2147483647. This period is limited by the constraints of the 'long' data type in C++ but is sufficient for the application to be evaluated against the stated goals in section 1.5, albeit with a degree of repetition of seeds (in this case approximately 129 stars will share each unique seed). This is not a limitation of the framework or architecture and we could easily substitute this simple PRNG with another pseudorandom number generation algorithm using a larger data type.

Additionally, the following functions were implemented for the MyPRNG class for use in this implementation.

double GetNextScaled() This generates a pseudorandom number scaled between 0 and 1;

double GetNextSkewed(double min, double max, double factor) This generates a pseudorandom number between min and max which is weighted toward the min value. This is achieved using the formula:

$$X_r = \frac{X_n^{factor}}{X_{max}^{factor-1}}$$

where X_r is the resultant skewed pseudorandom number, X_n is a regularly generated pseudorandom number, X_{max} is the maximum value that X can have (the size of LONG_MAX in this case) and $factor$ is an arbitrary power used to control weighting.

4.2.4 Tree Navigation

The tree begins with a single root node which represents the galaxy as a whole. The user is provided with a user interface to enable them to navigate the tree. This is further explained in section 4.2.5. When the user selects the Generate button on the user interface the generate function for the currently selected node is called. If this node has not reached the termination point the node will be subdivided and the child nodes created according to the applicable IFeature (if the current node does not have an attached feature a search up the tree will be carried out until an IFeature is found). The user will then be presented with a choice of children to trace down the tree. When the user presses the Go Down button the selected child will become the currently active node in the tree. In the case that children have not yet been generated, this will result in a call to the Generate function first and the first child will become the current node.

Similarly, when the user pressed the Go Up button the parent of the current node will become the current node, moving back up the tree.

While navigating the tree up or down the tree will prune itself such that all that is kept in memory is a chain of parent-child nodes from the current node to the root node. By doing this it is ensured that there will only be a number of nodes in memory equal to the depth within the tree of the current node, unless the user has manually called for generation of child nodes at the current node. Likewise, these manually generated nodes can be removed by pressing the Prune button.

Addressing

The project also implements an addressing system which allows the rapid generation of a galaxy to any level of detail using a string based address. Every node in the tree is assigned a unique address upon creation. This address consists of a string containing the seed used for the root node of the galaxy followed by a series of numbers corresponding to the indices of the children that must be generated to reach the node with the specified address, delineated by full stops. For example the address 12345.1.2.3.4.5.6.7.1.2.3.4.5 refers to a leaf node in the galaxy with the seed 12345 (in this case the address refers a node with a single star, three planets and it can be inferred from the length of the address that the node's depth will be 12).

4.2.5 User Interface

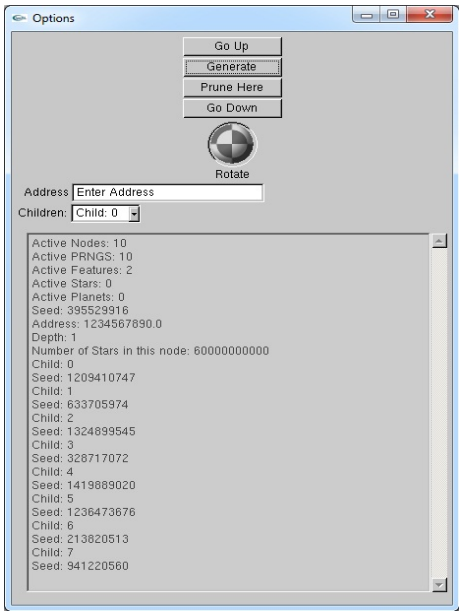


Figure 4.4: The user interface implemented for the demonstration project.

Fig. 4.4 shows the user interface created for this demonstration project. The user interface was created using GLUI User Interface Library [15]. The UI provides functionality for navigating the tree to any level of detail as well as for manipulating the visualisation of the galaxy and entering the address of a particular node. Furthermore,

this user interface provides information about the tree including the number of nodes, features, stars and planets which are currently in memory and the seed and address of the current node as well as the seeds for any children which have been generated for the current node.

4.2.6 Visualization

The galaxy is visualised by drawing points in 3D space in order to represent the distribution of stars according to the structure of the applicable IFeature. Clearly it is not practical to calculate or display the positions of three hundred billion stars so representative points must be generated heuristically. When a galaxy is created or the current node is changed, a call is made to the Draw function of the current node. This function will find the IFeature applicable to this node, searching up the tree if necessary and call the Draw function of that feature, passing itself as a parameter. The draw functions for each of the features implemented will be described in more detail below.

ClusterFeature In the case that the current node is at the same depth as the Cluster Feature the points generated will be randomly distributed about the centre point of the cluster at a distance d, where $0 \langle d \langle r$ and r is the radius of the cluster. The distance d will be selected using the GetNextSkewed() function described in section 4.2.3 ensuring that a higher density of points will be generated closer to the centre, reflecting the distribution of stars in a cluster. This can be seen in Fig. 4.5. In the case that the node is not at the same depth as the ClusterFeature the closest point of the current nodes volume to the centre of the cluster is calculated and a distribution is done from this point. The skewing of these points is affected by the difference in depth between the current node and the ClusterFeature such that the distribution of points becomes more even the further apart the two are and therefore the smaller a volume the current node is representing.

SprialArmFeature In the SpiralArm feature, structure is achieved using a series of points to create a spline in order to represent the curve of the spiral arm itself. When calculating the distribution of points throughout a given volume the start and end points of the spline are found within the volume of the current node. A point is chosen along the spline, skewed toward the start of the spline

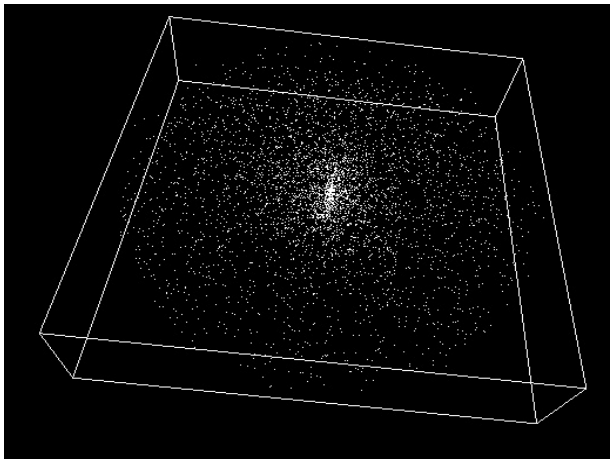


Figure 4.5: Visualization of the ClusterFeature.

in order to decrease the density of the stars towards the root of the arm. A second point is then chosen slightly further along the line and this is then used to calculate a vector in the direction of the spline at that point. This vector is turned 90 degrees in the plane of the galaxy, in a random direction and a new skewed number is generated in order to determine how far along this rotated vector the point will appear. Finally the point is displaced in the y direction, again skewed toward the spline. This gives a distribution of stars which is denser toward the root of the arm and denser the closer to the spline it appears as can be seen in Fig. 4.6. As the difference in depth between the current node and the SpiralArmFeature increases, a higher proportion of stars within the volume are placed purely randomly throughout the volume to reflect the loss of larger structure at smaller and smaller scales.

GalacticFeature In this application, the GalacticFeature is hard coded to represent the cluster and spiral arms below it in the tree using similar methods to those described for these feature above. The number of points to be drawn is split evenly between each of the features so they will all be readily apparent. The reason for doing this is that a realistic representation of the distribution of stars would heavily favour the bulge at the galaxy core and so a very large number of points would be required in order for the arms to be visible. The results of the

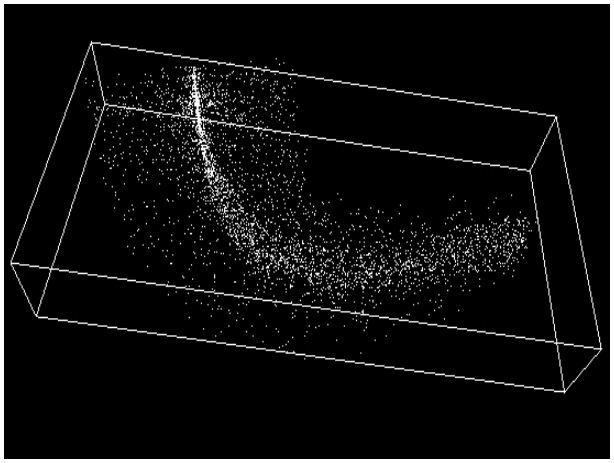


Figure 4.6: Visualization of the SpiralArmFeature.

GalacticFeature visualization can be seen in Fig. 4.7.

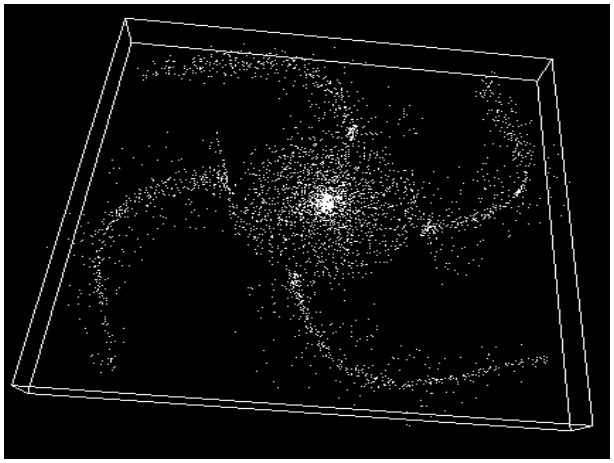


Figure 4.7: Visualization of the GalacticFeature.

Solar Systems As this application generates data down to the individual solar system level it was not practical to visualize these by the same method as other parts of the system for reasons of scale. As such, the SSNode class has a different implementation of the Draw() method which handles the drawing for solar systems directly rather than through the IFeature as in other cases. This draw function is a simple visualization of the stars and their planets (if the planets

have been generated) in this solar system. This drawing is not to scale but does represent the differences in scale based on the star or planets mass. An example of this visualization can be seen in Fig. 4.8.

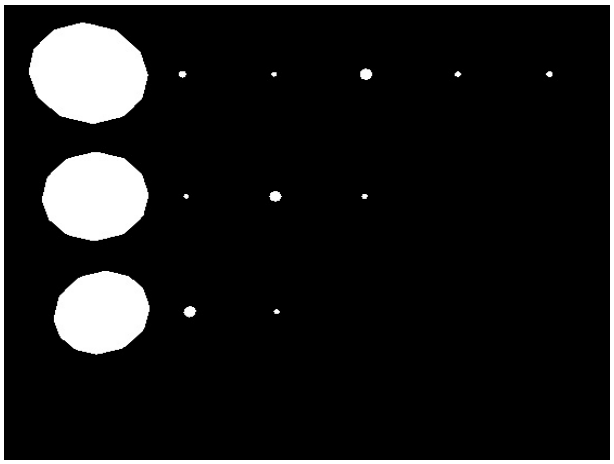


Figure 4.8: Visualization of a Solar System.

4.2.7 Stars and Planets

The focus of this dissertation was the creation and maintenance of large scale game-worlds. However, it was necessary to create content at the bottom of the tree for evaluation purposes. As such, the project defines classes for stars and planets. Although this information is not used within the project it does reflect the kind of information which might be produced for use in a game such as mass, radius, temperature semi-major axis, inclination etc as well as information used for generating data and tree operations such as pointers to planets owned by a star, pointers to pseudorandom number generators and seeds.

Chapter 5

Evaluation

In this chapter the galaxy generator presented in Chapter 4 will be used to evaluate the framework and the underlying architecture presented in Chapter 3 against the Goals stated in 1.4. A number of different strategies for evaluation were required to do this.

Section 5.1 will demonstrate the proof of concept for the architecture's ability to create the structure as required by Goal 1. This will be done with the use of screenshots of the project in action demonstrating the usage of several features at different levels of the tree.

Section 5.2 will measure the memory and processor usage for this project. These measurements will be used to show how well the system meets the performance and scalability requirements of Goal 2 and be used to infer performance in a hypothetical computer game.

Section 5.3 will analyse the architectures generality and how well it can be adapted for use in other projects as required by Goal 3.

5.1 Visual Structure

This section demonstrates the proof of concept of the systems ability to heuristically generate visual data at any point in the tree which would be consistent with the larger system. In order to achieve this, a simple point drawing system was developed in order to represent the distribution of stars within the volume of space bounded by the current node on the tree as described in section 4.2.6.

Chapter 4 explains how the system uses a series of classes derived from the IFeature interface which handle these heuristics for nodes at that depth or below. For this evaluation the IFeature classes that will be used are the GalacticFeature class, the ClusterFeature class and the SpiralArmFeatureClass.

5.1.1 GalacticFeature

The GalacticFeature class has functionality to draw a representation of the galaxy as a whole, in this case a central cluster to represent the galactic bulge and four spiral arms radiating out from this. In this implementation the shape of the galaxy is fixed to represent these substructures but uses similar code for drawing as these substructures for consistency. As each child node of the root node of the tree will itself have an IFeature implementation there is no requirement for the GalacticFeature class to estimate structure at lower levels of the tree, this will be handled by those features in the child nodes. The resulting galaxy produced can be seen in Fig. 5.1.

5.1.2 ClusterFeature

The ClusterFeature class represents a globular cluster of stars. This is a roughly spherical collection of stars with increasing density toward the centre of the cluster. The ClusterFeature distributes the points which represent the stars randomly about this centre point, weighting the random numbers towards the centre. At levels of the tree below the level of the ClusterFeature, this distribution from a central point is still apparent but as the difference in levels increases the weighting of the random points placed is reduced to show that at smaller and smaller scales this structure will be less apparent and the stars will seem more evenly distributed. This can been seen in Fig. 5.3 where (a) shows the visualization at the same level of the tree as the ClusterFeature, (b) is the representation at one level below the feature and (c) shows the distribution of stars at five levels below the feature and so the weighting of the distribution toward the centre (the upper left corner in this case) is significantly reduced.

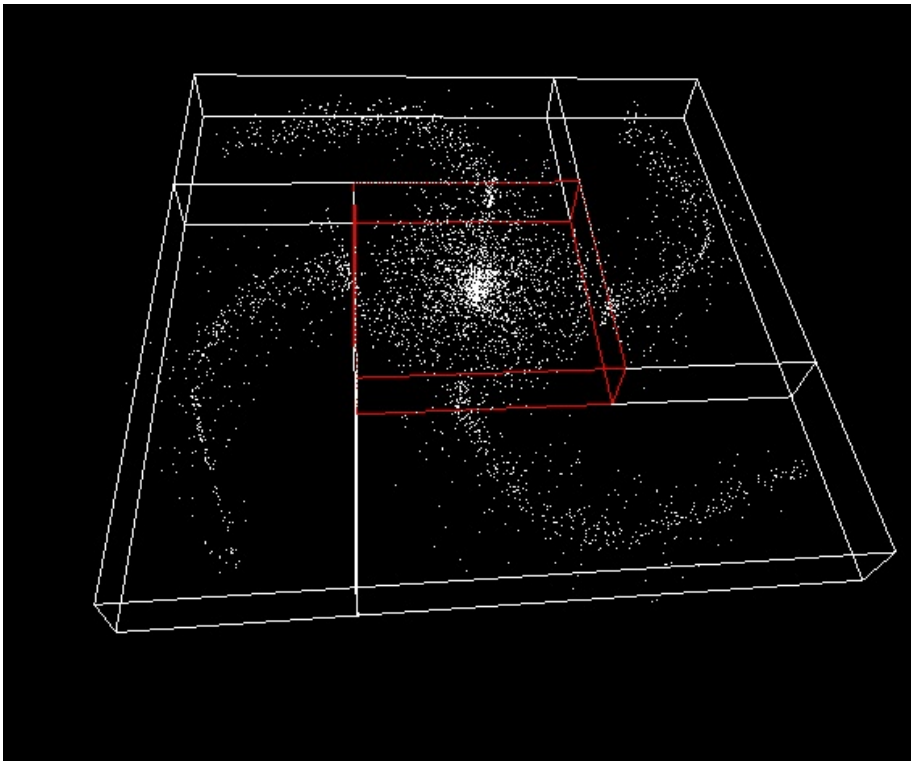


Figure 5.1: A galaxy with central cluster and four spiral arms.

5.1.3 SpiralArmFeature

A spiral arm is a region of stars which extends in a spiral fashion from the central bulge of a galaxy and which tapers as the distance from the centre increases. They are a common feature of both spiral and barred-spiral galaxies. In the SpiralArmFeature class this shape is represented through the use of a spline. Points to represent stars are placed randomly within the volume but weighted to be denser towards the root of the arm and denser from the spline outwards in order to mimic the structure of a spiral arm. At the tree depth where the SpiralArmFeature appears, these points are distributed along the entire length of the spline. At depths below that of the feature, this distribution occurs only along the portion of the spline which intersects the current nodes volume. Additionally as this difference in depth increases a proportionately larger amount of points are placed randomly throughout the volume to show the apparent loss of structure at smaller and smaller scales. The results of this can be seen

in Fig. 5.4. Fig 5.4 (a) shows the representation at the same depth of the tree as the SpiralArmFeature, (b) shows the distribution at one level removed from the feature and (c) shows the distribution at two levels removed where the loss of structure is becoming apparent.

5.1.4 Visual Structure Summary

As has been shown in the preceding sections, the project and the underlying architecture do meet Goal 1's requirement to impose structure at different levels of the tree. However, some limitations of both the project and the architecture are apparent.

Firstly, the heuristically generated points are unique to that level of the tree. This means that smooth transitions between different levels of detail are not provided by this project, resulting in some 'popping' as the tree is navigated. This is a limitation of the project itself rather than the architecture and could be addressed in another system.

Secondly, the architecture is limited to using only a single feature to generate structure for a given node on the tree. This makes it difficult to have the blending of two or more structures within a gameworld. In this case a kind of desirable blending would be along the border between the central cluster and the spiral arms. As is, the architecture is limited to situations where structures can be wholly contained in a single volume without overlap.

5.2 Performance

Goal 2 of this dissertation requires that the architecture provides a framework which will minimize the usage of both memory and processor time while being able to support the creation of large scale game worlds. To evaluate its performance in this respect, the demonstration application will be measured by these criteria and the architectures performance in a hypothetical computer game will be inferred.

5.2.1 Memory Usage and Scalability

Table 5.1 shows the memory usage of the classes used for this implementation. The project also produces 300 billions stars with the assumption that any given star will

Class	Size (bytes)
MyNode	160
SSNode	120
MyTree	1
ClusterFeature	8
GalacticFeature	80
SpiralArmFeature	16
MyPRNG	12
Star	56
Planet	64

Table 5.1: Memory usage for classes in the implementation.

have an average of 5 planets about this. From this it can be said that a naive approach to producing a similar galaxy, where all these stars and planets are precalculated and stored in memory would require approximately 102TB's of memory.

By comparison, if it is assumed that the distribution of stars is even throughout the galaxy, the subdivision of the tree will end at a depth of 11, with 55 stars active. This will result in a total of 12 MyNodes, 27 SSNodes (assuming an average of 2 stars per solar system), 95 MyPRNG's, 2 Features, and 55 Stars active in memory. From the table we can calculate that the memory usage at this point would be 9.25kB.

To generate a single solar system assuming it contained 2 stars with 5 planets each would require 13 MyNodes, 1 SSNode, 15 MyPRNGs, 2 Stars and 10 Planets and 2 features for a total of 3.05kB.

Assuming a non-uniform distribution of stars, as would be the case in a realistic representation of a galaxy, in the worst case a single node will contain 400 solar systems each with a single star. This worse case will still require 12 MyNodes, 400 SSNodes, 412 MyPRNGs, 400 stars and 2 features in memory for a total of 75.47kB.

In a hypothetical test case where each leaf node on the tree contained 400 stars as in the case above we can estimate that if the depth of the tree reached 22, the tree would be capable of supporting 29.5×10^{21} stars, more than the number of stars in the universe which is estimated to be 9×10^{21} and doing so would require approximately 56kB to store the tree down to an individual leaf node.

Finally, the graph in Fig. 5.2 shows the scalability of the framework by plotting the memory usage of this project against that of a naive approach. In this case the

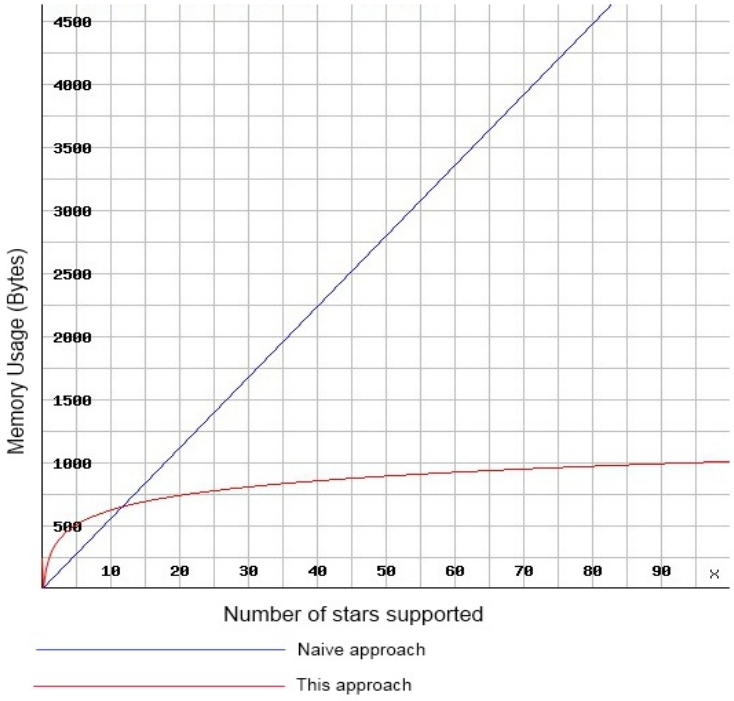


Figure 5.2: A graph showing the memory usage of this framework against that of a naive approach as the scale of the gameworld increases.

naive approach would be to create each star individually, at 56 bytes per star. This results in a linear increase of memory usage. By comparison, the method used in this dissertation achieves far better results. In this case the curve was calculated by adding the initial overhead of maintaining the tree and feature objects and then assuming a growth pattern in which each leaf node contains a single solar system which, in turn contains a single star at a total cost of 348 bytes per star. Even when using this worst case scenario for tree subdivision it can be seen that the tree framework used for this project will become more memory efficient than the naive approach when the gameworld grows beyond 13 stars.

Operation	Time Taken (milliseconds)
Moving down a level of the tree	$\langle$ 1
Moving up a level of the tree	$\langle$ 1
Pruning the tree	$\langle$ 1
Generate a galaxy from address	16
Generate visualization for GalacticFeature	31
Generate visualization for ClusterFeature	$\langle$ 1
Generate visualization for SpiralArmFeature	62

Table 5.2: Execution time for common operations.

5.2.2 CPU Usage

The test machine used to capture the results shown in Table 5.2 was a Windows 7 PC with a 2.1GHz dual core processor and 3GB of ram. Table 5.2 shows the execution time in milliseconds for some of the common operations involved in this implementation. In many cases the execution time was too small to be accurately measured being less than 1 millisecond. A brief description of what is involved in these operations follows.

Moving down a level of the tree This operation requires the children to be generated for the current node, seeds for them generated and assigned and the space divided amongst them. Following this a child is selected as the new current node and the other children are removed from memory. This operation assumes that children had not already been generated.

Moving up a level of the tree This operation involves moving the current node to point at its parent and then pruning all information below this point on the tree.

Pruning the tree This operation involves removing all data below the current node on the tree. This can be other nodes of type MyNode, nodes of type SSNode or Planets about stars. In all cases execution time was less then 1 millisecond.

Generating a galaxy from an address In this case an address was supplied for the leaf node of a galaxy. The galaxy was generated and the tree traversed to the leaf node in an average of the time indicated.

Generate visualization for GalacticFeature This operation represents the time taken to generate the 4000 points used to visualize the galaxy as a whole.

Generate visualization for ClusterFeature This operation represents the time taken to generate the 4000 points used to visualize the cluster at any level of the tree.

Generate visualization for SpiralArmFeature This operation represents the time taken to generate the 4000 points used to visualize the spiral arm at the highest level. As the tree is subdivided down from this level the execution time drops significantly, falling below 1ms at 3 levels below that at which the feature appears.

From these figures we can now examine the costs of some of the actions which may be common place in a computer game implementation of this project.

Travelling It would almost certainly be required that a player would require the ability to travel between star systems in any game implementation. As the player moves between the volumes described by a single node, it is necessary to step back up the tree a sufficient amount to generate the neighbouring volume. In a worst case scenario, the entire tree must be walked up and back down again if the player were to pass through the centre of the galaxy. However the cost of stepping up are down the tree is less than 1 ms. In this worst case scenario, this would require 24 such actions to walk the entire tree and so would still take less then 24ms.

Teleporting A common gameplay feature in space based games is the ability to teleport to different parts of the gameworld. This action could be represented using the address finding scheme implemented for this project. In this case the player could teleport to any part of the gameworld in 16ms.

Searching Another desirable feature in a computer game maybe the ability to search for a particular kind of star or planet. This is a function not addressed by this architecture and so a brute force search would be necessary.

5.2.3 Performance Summary

The figures from the preceding sections show that the architecture and the galaxy generator based upon it meet the requirements stated by Goal 2. The architecture has been shown to be able to support very large gameworlds and to do so with minimum cpu and memory usage although it does not reflect all the requirements that a game implementation may require of it.

5.3 Generality

The last of the goals stated in section 1.4 was that of creating a software architecture and accompanying framework which were general and flexible enough to be used in a variety of applications. Generality is a difficult concept to measure. Ideally, a series of different applications would be developed using this framework to evaluate its performance in this way, however building more than one application was beyond the scope of this dissertation. It should be noted that the architecture is itself a generalization and extension of the approach described by Hahn et al. [1] for the procedural generation of buildings. The framework should therefore be capable of generating buildings in this fashion. Additionally, the application developed for this dissertation for the procedural generation of galaxies shows that the framework is flexible enough to be used in a very different context.

Finally, it should be noted that the architecture itself is language and platform independent and an implementation of the architecture could easily be created for another system

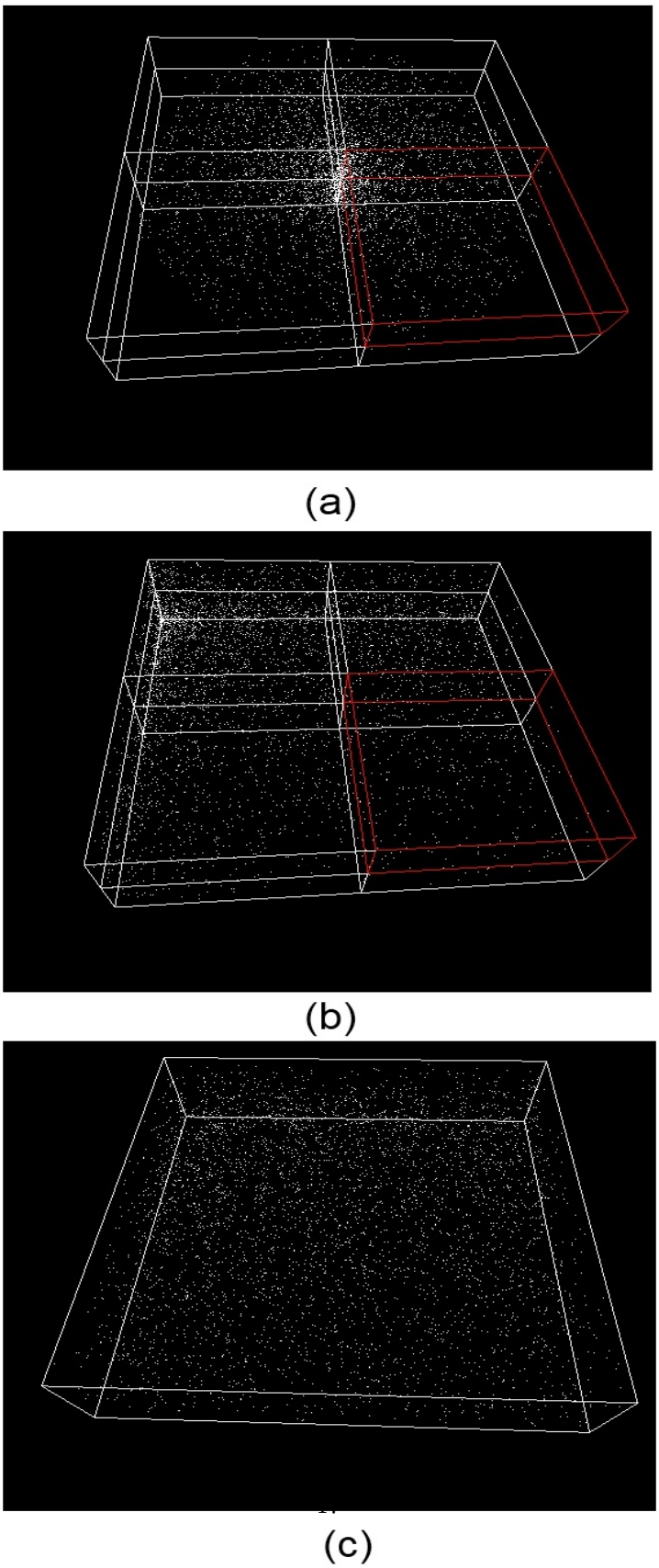
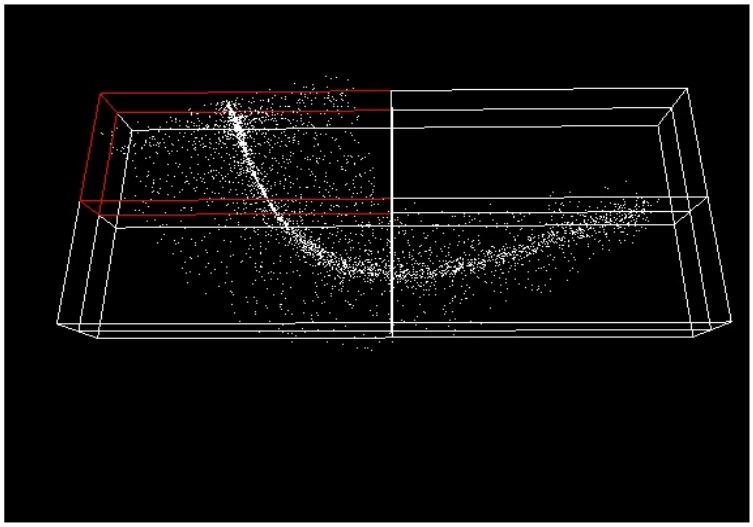
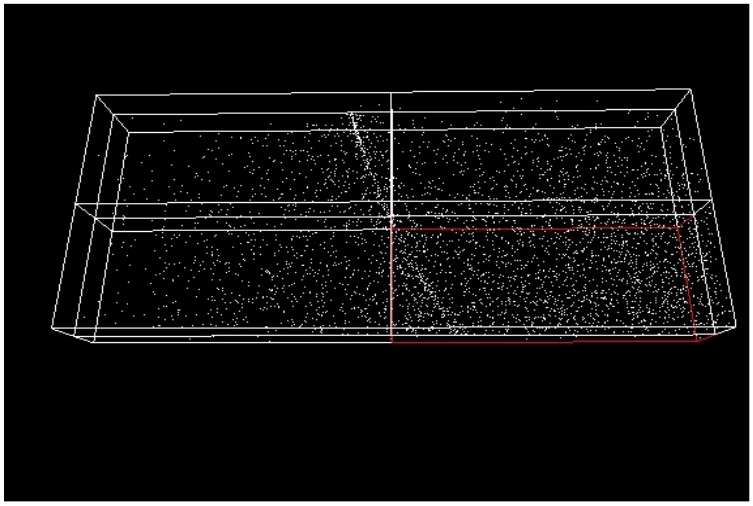


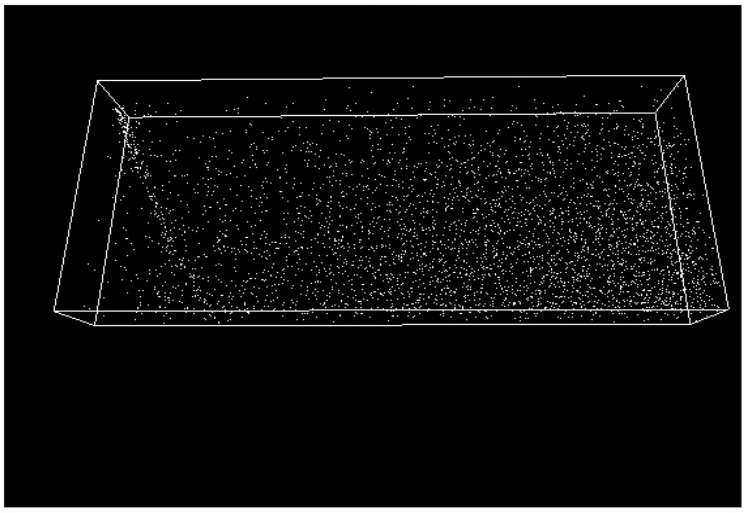
Figure 5.3: The estimation of structure at different levels of the tree for the Cluster-Feature. (a) is at level 1, (b) is at level 2 and (c) is at level 3.



(a)



(b)



(c)

Figure 5.4: The estimation of structure at different levels of the tree for the SpiralArm-Feature. (a) is at level 1, (b) is at level 2 and (c) is at level 5

Chapter 6

Conclusions

In Chapter 1 we saw that procedural content generation is popular through the games industry for meeting the increasing high requirements for detail and fidelity in modern gameworlds. Despite this there is still a lack of standardization of structure and logic for procedural content generation and this is what this dissertation sought to address. Several challenges to such a task were also identified, namely, the ability to impose structure upon procedurally generated content, meeting the severe hardware utilization requirements required for games and the need to standardize the approaches used for this type of content.

A selection of the literature available in this field was reviewed to assess some of the methods and practices used for procedural content generation and their ability to address the challenges described above. From this analysis a software architecture was proposed and presented which was intended to provide a system for standardizing the structural logic for procedural generation in a manner which was still flexible and general enough to be integrated with other applications, specifically with game applications.

To evaluate this architecture and the framework implemented from it, an application was developed to procedurally generate realistically sized galaxies. This would demonstrate the flexibility, scalability and generation logic of the architecture.

This application was then evaluated for its ability to overcome the stated challenges and meet the stated goals. Its memory and processor usage were measured to demonstrate its ability to be integrated in a resource scarce environment and its ability to

handle the kind of functionality which would be desirable in a game. The ability to create structure at multiple levels of detail was visually demonstrated and the generality of the architecture and framework were critically examined.

6.1 Future Work

There remain several limitations for this architecture and challenges to procedural generation which remain unaddressed in this dissertation. The following sections outline a selection of these which may be investigated in the future.

6.1.1 Time

The challenge of creating procedural content which evolves over time is not something which is planned for in the presented architecture. It is made especially challenging in this context as the architecture assumes that any structural aspect of the gameworld may be wholly contained by a single feature object. If there is the requirement that structural features merge, separate or overlap over time this will present great difficulty to the architecture as is. This difficulty is compounded by the application's use of spatial subdivision techniques for the splitting of the gameworld into discrete volumes. In a changing system, the tree would need to be continuously regenerated over time to reflect changes in the gameworld and this is clearly not ideal. It may be worth investigating whether or not the architecture could be adapted in order to meet this temporal challenge and be used for procedurally generating gameworlds which evolve over time.

6.1.2 Popping

As shown in 5.1 the application developed for this dissertation heuristically generated visual data to represent the structure of the galaxy. This was limited to creating a new visualization for each node of the tree independently of any other and as a result there was a lack of continuity when transitioning from one level of detail to another. As the selected node of the tree was changed there was distinct 'popping' of the visualization. That is to say that it was apparent to the user when new visualizations

were being generated and that these visualizations were only loosely related to one another. Implementing a system which could address this issue could be explored.

6.1.3 Feature Merging

A limitation of the architecture is that each node may only have one feature applied to it at any given time. This works well when the structures of the gameworld are well delineated, such as in the case of a building were rooms are clearly separate from each other. However, in a more organic setting, like a landscape generator, this makes the blending of regions problematic. The intuitive response would be to investigate whether it is feasible to adapt this architecture such that a node could be subject to multiple features at once, and whether the influence of features could be weighted over the volume of a node to create smooth transitions between features.

6.1.4 Integration

This architecture was designed and implemented with a view to being extensible. It would be worth attempting to combine this application for the procedural generation of galaxies with other applications, using the presented architecture, in order to create a higher fidelity large scale gameworld. A good candidate would be the project for the procedural generation of planets created by Hayes-McCoy [16].

Appendix

Attached is a disc containing the source code for the Framework and Galaxy Generator implemented for this dissertation.

Bibliography

[1] E. Hahn, P. Bose, and A. Whitehead, "Persistent realtime building interior generation," in *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, Sandbox '06, (New York, NY, USA), pp. 179–186, ACM, 2006.

[2] M. Nitsche, C. Asmore, W. Hankinson, R. Fitzpatrick, J. Kelly, and K. Margenau, "Designing procedural game spaces: A case study," in *Proceedings of FuturePlay 2006*, 2006.

[3] J. Zhang and J. Xu, "Optimizing octree motion representation for 3d animation," in *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, (New York, NY, USA), pp. 50–55, ACM, 2006.

[4] D. Braben and I. Bell, *Elite*. Firebird, Acornsoft and Imagineer, 1984.

[5] M. Persson and J. Bergensten, "Minecraft," in *http://www.minecraft.net*.

[6] J. Lluch, E. Camahort, and R. Vivó, "Procedural multiresolution for plant and tree rendering," in *Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, AFRIGRAPH '03, (New York, NY, USA), pp. 31–38, ACM, 2003.

[7] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," vol. 22, pp. 669–677, july 2003. Proceeding.

[8] M. Larive and V. Gaildrat, "Wall grammar for building generation," in *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, GRAPHITE '06, (New York, NY, USA), pp. 429–437, ACM, 2006.

[9] D. Ebert, *Texturingand Modeling: A Procedural Approach.* Morgan Kaufmann Pub, 2003.

[10] R. Marshall, R. Wilson, and W. Carlson, "Procedure models for generating three-dimensional terrain," in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '80, (New York, NY, USA), pp. 154–162, ACM, 1980.

[11] M. Danaher, "Dynamic landscape generation using page management," in *the 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2002*, pp. 135–138, Citeseer, 2002.

[12] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time procedural generation of 'pseudo infinite' cities," in *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, (New York, NY, USA), pp. 87–ff, ACM, 2003.

[13] T. Wang, "Integer hash function," in *http://www.concentric.net/ ttwang/tech/inthash.htm*, Jan. 1997.

[14] R. Gardner, "longrand," in *http://www8.cs.umu.se/ isak/Snippets/rg_rand.c*, June 2011.

[15] P. Rademacher, "Glui user interface library," in *http://www.cs.unc.edu/ rademach/glui/*, June 2011.

[16] D. Hayes-McCoy, "Procedural generation of planetary bodies," Master's thesis, University of Dublin, Trinity College, 2011.