

**AN OBJECT-ORIENTED DESIGN
FOR HIERARCHICAL B-SPLINE SURFACES**

By

Hailin Yan

B.Sc University of Science and Technology of China

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE**

in

**THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE**

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

February 1993

© Hailin Yan, 1993

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date March 15, 1993

Abstract

This thesis documents an object-oriented software system that supports free-form surface modelling based on the hierarchical overlay methodology of [Forsey88]. This work is motivated by the need to provide a space-efficient representation of tensor-product hierarchical spline surfaces, multiple offset method support, and general surface representation. This design uses a spatial data structure, the quadtree, to achieve this goal. The quadtree, itself a hierarchical data structure, is very suitable in this application because of its ability to focus on the interesting subsets of the data. This focusing results in an efficient representation. The quadtree is also attractive because of its conceptual clarity and ease of maintenance.

The package is implemented in *C++* to provide: a) extensibility so that the new tools can be easily integrated into the existing package; b) reusability of code; and c) localization of code. Finally, this object-oriented hierarchical design keeps all of the original features of the hierarchical overlay method of [Forsey90].

Table of Contents

Abstract	ii
List of Tables	v
List of Figures	vi
Acknowledgements	viii
1 Introduction	1
1.1 The Problems and Motivations	1
1.2 The Design Goals	3
1.3 Overview	4
2 Background	6
2.1 The Hierarchical Surface Methodology	6
2.2 Object-Oriented Programming and the Waterloo <i>C++</i> Spline Classes . .	11
3 Design and Implementation	19
3.1 Analysis of Hierarchical Surface Representations	19
3.2 Overview of Functional Relationships and Structures in the Design	27
3.3 Functional Specification for Each Class	35
3.3.1 Hierarchical Overlay Classes	35
3.3.2 Quadtree Classes	43
3.3.3 The Support Classes	51

3.4	Design Review	58
4	Storage and Performance Analysis	59
4.1	Storage Benchmarks	59
4.1.1	Storage Analysis in the Single-overlay Quadtree Representation .	60
4.1.2	Storage Analysis in the Multiple-overlay Quadtree Representation	61
4.2	Performance Benchmarks	65
4.2.1	CV Navigation	66
4.2.2	Traversals in a Hierarchical Surface	70
4.2.3	Evaluation	71
4.2.4	Refinement	72
4.2.5	Multi-level Editing	73
5	Conclusion and Future Work	75
5.1	Conclusion	75
5.2	Future Work	76
	Appendices	79
A	Mathematical Background on Tensor-product B-spline Surfaces	79
B	The Definition of Quadtrees and Their Characteristics	84
	Bibliography	87

List of Tables

3.1	Four kinds of quadtree data structures for hierarchical overlays	24
4.2	Storage overheads for a fully refined surface in three kinds of data structures	62
4.3	Storage overheads for a sparsely refined surface in three kinds of structures	63
4.4	Execution time for getting a CV in the <i>multiple-overlay quadtree</i>	67
4.5	Execution time for getting a CV in <i>single-overlay quadtrees</i>	67
4.6	Execution time (in μsec) for getting a CV and its neighbour in the <i>multiple- overlay quadtree</i> representation	68
4.7	Execution time (in μsec) for getting a CV and its neighbour in the <i>single- overlay quadtree</i> representation	69
4.8	Execution time (in μsec) for getting a CV and the CV's in its <i>parent/child patch</i> in the <i>multiple-overlay quadtree</i> representation	69

List of Figures

2.1	Surface $S^{[k]}$ and edited surface $S^{[k+1]}$	8
2.2	A 16-patch surface described by 49 control vertices	9
2.3	A 16-patch surface with refinement and overlay control vertices	10
2.4	Basis Class Hierarchy	14
2.5	The hierarchy for parametric tensor-product surfaces	17
2.6	The basis spline surface object	18
3.7	A hierarchical surface and its <i>multiple-overlay quadtree</i> representation . .	25
3.8	A hierarchical surface and its <i>single-overlay quadtree</i> representation . . .	26
3.9	A hierarchical bi-cubic B-spline surface	28
3.10	The initial hierarchical surface in a <i>multiple-overlay quadtree</i>	30
3.11	The hierarchical surface in a <i>multiple-overlay quadtree</i> after one refinement	31
3.12	The hierarchical surface in a <i>multiple-overlay quadtree</i> after the second refinement	32
3.13	The final hierarchical surface in a <i>multiple-overlay quadtree</i>	33
3.14	The initial hierarchical surface in <i>single-overlay quadtrees</i>	34
3.15	The hierarchical surface in <i>single-overlay quadtrees</i> after one refinement .	36
3.16	The hierarchical surface in <i>single-overlay quadtrees</i> after the second refine- ment	37
3.17	The final hierarchical surface in <i>single-overlay quadtrees</i>	38
3.18	A hierarchical B-spline surface object in the <i>multiple-overlay quadtree</i> . .	38

3.19	A hierarchical B-spline surface object in <i>single-overlay quadtrees</i>	39
3.20	HierOverlay Class Structure	39
3.21	QuadTree Class Hierarchy	44
3.22	The hierarchy for a control node definition	51
3.23	The Control Node Matrix Structure	57
4.24	The definition for the n^{th} lowest level in a quadtree.	64
5.25	One non-uniform refinement case in the <i>single-overlay quadtree</i> structure	77
A.26	Regions for four equal patches	82
B.27	A region, its maximal blocks, and the corresponding quadtree. (a) Region. (b) Block decomposition of the region in (a). (c) Quadtree representation of the blocks in (b).	85

Acknowledgements

I would like to thank Dr. David Forsey, my thesis supervisor, for his guidance and encouragement throughout my work on this thesis. The discussion between us improved the content and presentation of this thesis significantly. I would also like to thank Dr. Jack Snoeyink for reading through the draft of this thesis, his comments and his help. Many thanks go to the graduate students of the Department of Computer Science, especially Christopher Healey, Karen Kuder, Gene Lee, Stanley Jang, Vishwa Ranjan and Tony Lau for their help and suggestions. I also wish to thank those people from the Waterloo Computer Graphics Laboratory for supplying the *C++* Spline Software Package, which I made use of in my implementation.

Chapter 1

Introduction

This thesis is motivated by the need for an efficient representation of hierarchical B-spline surfaces. This chapter describes the problems inherent in constructing free-form hierarchical B-spline surfaces, briefly presents our design goals, and closes with a thesis overview.

In this thesis, it is assumed that the reader has a basic knowledge of splines as used in the design of surfaces in computer graphics, at least to the level presented in Appendix A. For more information, refer to [Bartels87] or [Farin90].

1.1 The Problems and Motivations

In the traditional approach to B-spline surface modelling, a surface is defined by: an $m \times n$ matrix of control vertices, two knot vectors determining the location of each CV¹ in parametric space, and the appropriate basis functions (See Appendix A). The number of patches in the surface is increased from mn to $(m + 1)n$ or $m(n + 1)$, using a process called *knot insertion* or refinement. A general, space-efficient approach for representing tensor-product spline surfaces was proposed in [Forsey90]. It offers greater editing flexibility than is normally found in systems using traditional representations ([Riesenfeld81], [Tiller83], [Maiorino85] and [Sederberg & Parry86]). Forsey introduced a data structuring technique which allows *local refinement* of tensor-product spline surfaces (such as

¹CV means control vertex.

B-splines and their rational counterparts) so that the number of patches in a given region can be increased without affecting the rest of the surface. Local refinement, coupled with a *reference-plus-offset* representation of the hierarchy, allows surface manipulation independent of refinement by exploring the local geometry of the surface. This “hierarchical form” is applicable to any spline with a refinement procedure and locally supported basis functions. These properties can be used in the task of creating a complex shape from a single, continuous, tensor-product spline surface, and are useful in free-form surface design in general. For details, refer to Section 2.1.

Although the above hierarchical surface approach overcomes some of the shortcomings of traditional tensor-product spline surfaces, the current prototypical hierarchical B-spline editor exhibits the following cumbersome characteristics:

- A flexible, but space-inefficient internal representation, in which each control vertex uses 6 pointers (4 pointers to its immediate neighbours in the CV matrix in the same overlay and 1 each to a corresponding CV in the two adjacent levels);
- Only bi-cubic B-splines are available;
- The process of adding new offset methods is not structured;
- Multiple hierarchical surfaces cannot be manipulated within a single invocation of the editor.

This thesis is the result of a re-evaluation of the software needs and requirements of a hierarchical spline editing system, motivated by:

1. Extensibility. The package must not be limited to one particular type of spline. It is essential that it is easy to develop new surface editors which can be used with a variety of spline formulations.

2. Reusability of code. It is desirable to avoid repetition of code with similar functionality. Code should be reused rather than duplicated.
3. Localization of code. Code that deals with related matters, or structurally similar algorithms to achieve related goals, should be concentrated, as much as possible, in a single location.
4. More compact representation of the hierarchical surface itself.

To provide a solid foundation, a *C++* spline software system supporting interactive spline modelling was obtained from the University of Waterloo's Computer Graphics Laboratory ([Bartels91]). This system contains a reusable collection of support programs for splines, rendering, refinement, display and interaction.

1.2 The Design Goals

The ultimate purpose of this research is to aid in the development of a highly interactive and intuitive modelling tool to replace the existing prototypical editor. This thesis details the implementation of *C++* tools that both efficiently represent the hierarchical surface and provide a complete set of procedures to deal with the manipulation and instantiation of such a surface.

A hierarchical surface is comprised of multiple levels of "spline overlay" with the parametric spacing of patches in each overlay being half of that in the preceding level. In this multi-resolution representation of a surface, level 0 has the largest parametric spacing (typically 1.0) and level $n + 1$ has a spacing that is half of that of level n .

The criteria for choosing an efficient data structure to represent a hierarchical surface are:

- fast access to an arbitrary control vertex in the hierarchy

- fast access to the immediate neighbours of a given control vertex
- fast access to level $n \pm 1$ from level n
- fast traversal of the hierarchy
- minimized storage requirements

An efficient data structure for a hierarchical surface representation is desired. This data structure must be easy to implement, minimize memory requirements and be flexible enough to support future extensions. Since our design adopts the hierarchical overlay method, it is not surprising that a data structure has been chosen with a similar hierarchical nature. That is, a hierarchical data structure, the *quadtree*, is used to represent a hierarchical spline surface.

1.3 Overview

The rest of this thesis is organized as follows:

Chapter 2 describes the hierarchical spline methodology, object-oriented programming and the Waterloo *C++* spline classes.

Chapter 3 gives a detailed description of our design of an object-oriented hierarchical B-spline surface. We capture the major characteristics of our design using an example, and then explore their advantages.

Chapter 4 presents an assessment of the design with respect to the space- and time-efficiency of the main operations in our surface modeller. We then analyze the strengths and weaknesses of the design.

Chapter 5 concludes the thesis with the results of our research and presents directions for future study.

Appendix A gives B-spline and tensor-product B-spline surface definitions.

Appendix B describes a hierarchical data structure, the quadtree, and its characteristics.

Chapter 2

Background

2.1 The Hierarchical Surface Methodology

¹ Tensor-product B-splines are flexible surface representations, but they possess a deficiency when it comes to refinement. Refinement is usually advocated as a means of gaining finer control over a spline surface during editing. However, refinement may add more control vertices than required. One method of localizing the effect of refinement is the hierarchical surface representation presented in [Forsey90].

A degree (k, l) tensor-product B-spline surface has the form:

$$S(u, v) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \vec{V}_{i,j} B_{i,k}(u) B_{j,l}(v)$$

The control vertices $\vec{V}_{i,j}$ are arranged in an $m \times n$ rectangular array, called the control vertex mesh indexed by i, j . Each $\vec{V}_{i,j}$ is a vector (Here, vector refers to the cardinal coordinates). $B_{i,k}(u)$ and $B_{j,l}(v)$ are the univariate B-spline basis functions. The variables k and l are the orders of the B-splines in the u and v parametric directions, respectively.

Hierarchical surface representation is a data structuring technique imposed upon a tensor-product spline surface. This data structure relies on two properties of the B-spline basis function: *local support* and *refinement*. Local support means that the area on a surface affected by a single control vertex is bounded. The procedure of refinement produces an exact re-representation of the original surface but with a larger number of

¹This section is referenced from [Forsey90].

patches and control vertices. Local refinement exploits the local support property of the B-spline basis function to restrict the extent of refinement over the surface by increasing the number of patches in a restricted region.

A level - k surface $S^{[k]}(u, v)$ is defined over *control nodes*² $\vec{V}_{i,j}^{[k]}$ by the following equation:

$$S^{[k]}(u, v) = \sum_i \sum_j \vec{V}_{i,j}^{[k]} B_{i,k}^{[k]}(u) B_{j,l}^{[k]}(v)$$

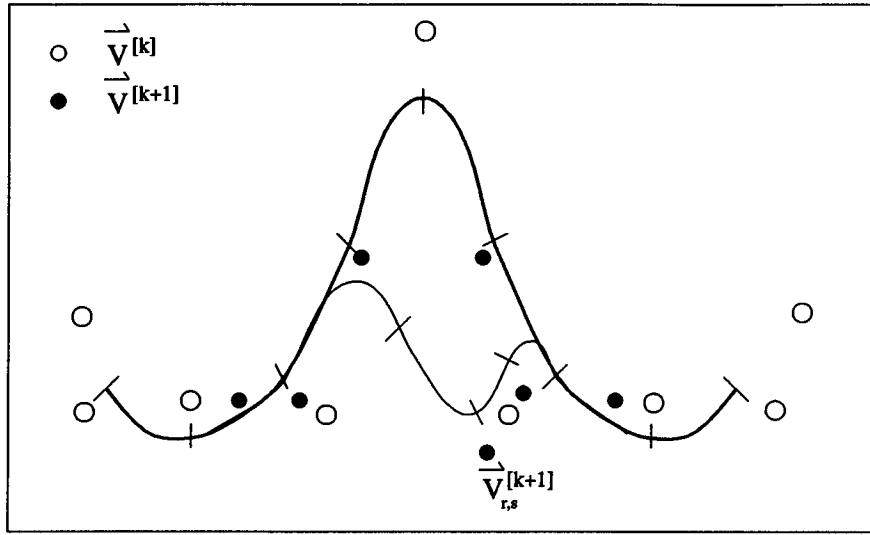
The surface derived from the refinement of a restricted region of $S^{[k]}$, defined by a subset of the $\vec{V}^{[k]}$, is the level - $k + 1$ surface

$$S^{[k+1]}(u, v) = \sum_i \sum_j \vec{V}_{i,j}^{[k+1]} B_{i,k}^{[k+1]}(u) B_{j,l}^{[k+1]}(v)$$

defined by control nodes $\vec{V}_{i,j}^{[k+1]}$ and a different set of basis functions obtained through refinement of the restricted region. Each $S^{[k]}$ is called an *overlay*. The level - k surface is the *parent* of the *child* at level - $k + 1$ and the level - 0 surface is called the *root-level*. Only midpoint refinement is used. Therefore, each child overlay has uniform knot vectors with half of the spacing of its parent overlay.

If one control node of $\vec{V}_{r,s}^{[k+1]}$ is moved (*edited*), the surface $S^{[k+1]}$ departs from its parent, $S^{[k]}$, in the local area influenced by that particular control node. If editing is restricted to those control nodes of $\vec{V}^{[k+1]}$ whose corresponding basis functions are zero at the boundary between the surfaces $S^{[k]}$ and $S^{[k+1]}$, discontinuities will not appear in the surface. Figure 2.1 shows the surface $S^{[k]}$ superimposed on the surface $S^{[k+1]}$ after the control vertex $\vec{V}_{r,s}^{[k+1]}$ has been moved. The resulting composite surface retains the continuity properties of the underlying basis functions because the nature of the basic surface representation has not changed.

²Any control vertex in any level of a hierarchical surface is called a control node. In the remainder of this thesis, CV means control node in a hierarchical surface.

Figure 2.1: Surface $S^{[k]}$ and edited surface $S^{[k+1]}$

Local refinement can be repeated on the interior of an overlay to create further overlays within overlays. The basic operation of creating an overlay consists of designating a control node $\vec{V}_{r,s}^{[k]}$ on the surface at a particular level of refinement and executing a refinement step to re-represent the area influenced by the refined control node $\vec{V}_{r,s}^{[k+1]}$. If this causes overlays to overlap each other in $S^{[k+1]}$, the overlapping overlays are made into a composite overlay by combining their respective control nodes into a single collection.

Figure 2.2 shows a schematic plan of 7×7 matrix of control vertices (circled x's), along with the 16 bi-cubic B-spline patches and the surface that they define. This constitutes the minimal portion of the surface that would change due to any movement of the central control node $\vec{V}_{r,s}^{[k]}$. Figure 2.3 illustrates the overlay that would result from local refinement. The black dots represent the control nodes in $\vec{V}^{[k+1]}$, and dashed boxes outline the smaller patches that the central control node $\vec{V}_{r,s}^{[k+1]}$ will influence. If only this central control node of $\vec{V}^{[k+1]}$ is moved, and all the others are held fixed, the patches given by the dashed boxes will remain an integral part of the surface and the boundary will remain continuous with the 12 surrounding patches defined by the control vertices

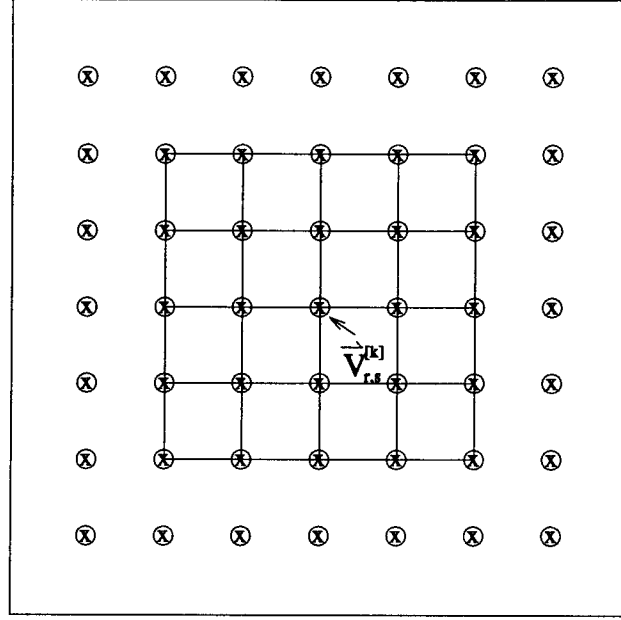


Figure 2.2: A 16-patch surface described by 49 control vertices

in $\vec{V}^{[k]}$. The degree of continuity at the boundary depends upon the continuity of the splines themselves. For bi-cubic patches, it will be C^2 ; while for bi-quadric patches, it will be C^1 .

For editing which involves the movement of several control vertices that modify a larger area of the surface, the new overlay must enclose the union of the individual single-vertex overlays. For editing that is to influence a smaller region, refinement will break each patch into several subpatches.

The net effect of repeated local refinement is a surface composed of a collection of overlays at different levels of refinement.

Two problems remain. Although editing the surface within an overlay cannot produce boundary anomalies (using the above definition of an overlay, the only control nodes that are *allowed* to move are those that will not affect the boundary), moving the surface immediately surrounding an overlay will cause tears to appear at the boundary between

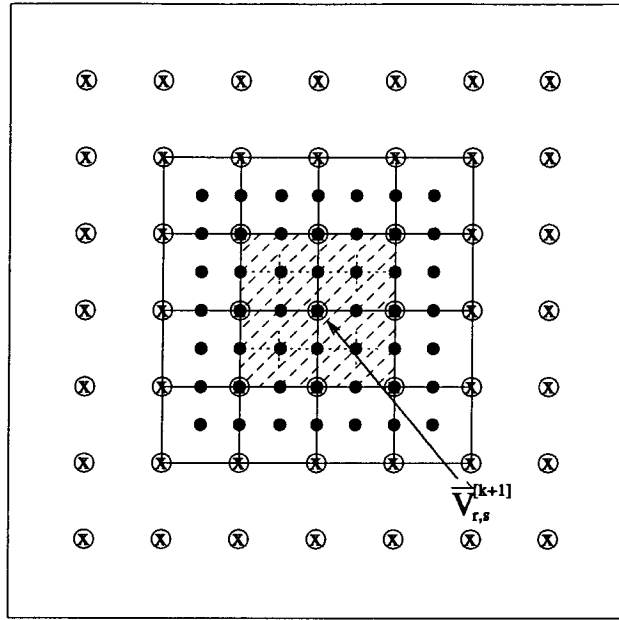


Figure 2.3: A 16-patch surface with refinement and overlay control vertices

patches defined at two different levels. Also, when editing takes place at one level of surface definition, any overlays resting within the edited area are expected to remain embedded in that area. The embedded overlays will follow editing changes only if they can be dynamically influenced by changes in the shape of some ancestor overlay. This amounts to saying that their control vertices must move in accordance with the movement of the section of the surface being edited.

These properties are achieved by a method called *offset-referencing*. The positions of the control nodes of any overlay are defined relative to a point on the parent surface $S^{[k]}$, rather than relative to a single fixed frame of reference defined by some external coordinate system. In this *reference-plus-offset* formulation, each control node $\vec{V}_{i,j}^{[k+1]}$ of any part of the surface, whether the root-level parent surface or an overlay at any level of refinement, is written in the form

$$\vec{V}_{i,j}^{[k+1]} = \vec{R}_{i,j}^{[k+1]} \oplus \vec{O}_{i,j}^{[k+1]}$$

where $\vec{R}_{i,j}^{[k+1]}$ is the *reference position*, and $\vec{O}_{i,j}^{[k+1]}$ is the *offset vector*. Both are specified separately for each control node $\vec{V}_{i,j}^{[k+1]}$ in each overlay. The position of $\vec{V}_{i,j}^{[k+1]}$ immediately after creating a new overlay using refinement is equal to $\vec{R}_{i,j}^{[k+1]}$ and, by definition, the value of $\vec{O}_{i,j}^{[k+1]}$ is zero. Any change to the position of a control node $\vec{V}_{i,j}^{[k+1]}$, is stored in the offset vector $\vec{O}_{i,j}^{[k+1]}$ as a relative change in position from the reference point $\vec{R}_{i,j}^{[k+1]}$. The operator \oplus specifies how $\vec{R}_{i,j}^{[k+1]}$ is combined with $\vec{O}_{i,j}^{[k+1]}$ and is called the *offset method*. This operator can be defined as one of several kinds of operations, such as vector addition, tangent plane, skeletal frame and dynamic function methods in our implementation.

Changes to the surface at any level of refinement close to the root level appear as global changes to the overlay surface at the current level of refinement. The current level of the surface has its control vertices modified in accordance with changes in the reference position information, and this in turn affects the reference information for finer levels of the surface. Likewise, adjustments to the offset affect the surface at the current level of refinement and all finer levels.

Rendering proceeds for any point on the surface using reference and offset information at the lowest level of the tree containing that point.

2.2 Object-Oriented Programming and the Waterloo C++ Spline Classes

This section briefly introduces the main concepts of object-oriented programming (*encapsulation, inheritance, polymorphism, overloading and genericity*) using examples from the Waterloo C++ Spline Software Package (WaSP).

The promise of object-oriented languages is that code written in an object-oriented fashion is highly modular, maintainable, flexible and reusable. These advantages arise from the virtues of *encapsulation, inheritance* and *polymorphism* manifested in various object-oriented languages in the form of *objects* defined by *classes*. A good introduction

to the basics of object-oriented programming may be found in [Meyer88]. For more details on the specific features of *C++*, consult [Ellis90, Lippman90, or Stroustrup91]. An informal explanation of these concepts in the context of the design philosophy for the WaSP spline classes will be presented.

Encapsulation is a fundamental aspect of object-oriented programming. Code and data are encapsulated when related data and algorithms are tightly bound together and can be accessed only through a well defined interface. This interface should be independent of the algorithms, data storage, and data management within the code. Encapsulated code provides services, but does not reveal the details of how those services are carried out.

Code and data that have been encapsulated are often referred to as an *abstract data type*. An abstract data type definition is called a *class* in *C++*. An instance of a class is called an *object*.

A class definition provides the template of an abstract data type, which includes *data*, *construction*, *destruction*, *access*, *management*, and *service*.

The following code segment shows part of the class definition for the class *NUBBasis* from WaSP:

```
// NUBBasis represents a non-uniform B-spline basis

class NUBBasis: public BBasis {

private:    // Private members can be accessed only by members
           // in this class or its friend classes/functions.

    NumberSequence knts; // knots.

public:    // Public members can be accessed by all the
           // instances of this class.

    // Constructor, set knts to [0,1,...].
    NUBBasis( int dimension, int order );

    // Destructor
    ~NUBBasis();
```

```
// Make knts a copy of ns.
void setKnots( const NumberSequence& ns );

// Add the value of a knot.
void addKnot( double knt, int mult=1 );

// Delete the value of a knot.
void deleteKnot( int i );

.....
}
```

Management and service are incorporated using code that defines the public interface in the form of *member functions*. The functions *setKnots*, *addKnot*, and *deleteKnot* are examples of member functions that provide various services to users of a *NUBBasis* object. *Member data* is also encapsulated in the class definition. Data can be declared to be either public or private. Public data can be accessed and modified directly by users of a *NUBBasis* object. Private data can only be accessed by the member functions of the given object. The variable *knts* is an example of a private data variable. Users of a *NUBBasis* object must “request service” through a call to a member function in order to access or modify private member data. Every new instantiation of an object gets its own unique copy of the member data variables in the class definition. Modifying a given *NUBBasis* object’s member data has no effect on the data values of other *NUBBasis* objects. This method of access control enforces encapsulation by making it impossible for a program using *NUBBasis* objects to depend on anything other than the public interface. In particular, the implementation of member functions can be changed freely, as long as the original public interface is maintained.

Initialization, allocation, and deallocation of objects and their corresponding member data are performed automatically through the use of special *constructor* and *destructor* member functions. The semantics of *C++* provide rules that specify when constructors and destructors are to be invoked. The function *NUBBasis(int dimension, int order)* is an example of a constructor for the *NUBBasis* class. The *C++* statement

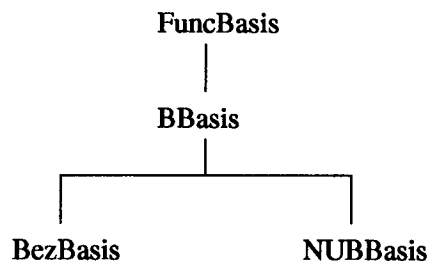


Figure 2.4: Basis Class Hierarchy

```
NUBBasis    nub_obj(8,4);
```

defines a new *NUBBasis* object called *nub_obj*, and automatically invokes the constructor function, which initializes the dimension and order of the object to 8 and 4 respectively. The destructor $\sim NUBBasis()$ is called whenever a *NUBBasis* object is destroyed either explicitly by an invocation of “delete” or implicitly upon the return of a procedure in which the object appears as a local variable. The destructor “cleans up” by freeing memory allocated to the object before it disappears.

Another fundamental concept in object-oriented programming is **inheritance**. Classes can be grouped into type hierarchies using inheritance. This feature is used to define a new class, called a *derived* class, in terms of an existing class, which is referred to as a *base* class. The derived class “inherits” the base class’ member functions and data. In addition to defining its own functions and data, the new class can replace “inherited” information, providing a natural supporting mechanism for incremental software development.

Inheritance also provides a logical way of organizing services into several layers of complexity, represented by classes in several layers of a hierarchy.³ Through the use of inheritance, the *C++* hierarchy for spline basis functions is organized in several layers (Figure 2.4).

³Reference from *C++ Primer*[Lippman91].

Class derivation permits code to be extended with a minimum of effort and disruption. For example, the member function *evaluate* is defined in the class *BBasis*, and thus it does not need to be redefined as a member function in the class *NUBBasis* because *NUBBasis* is a derived class of *BBasis*. Inheritance and class derivation provide the ability to *modify* services as well as to *augment* them. For example, there are no member functions *order()* and *knots()* in the class *FuncBasis*, but they do exist in the class *BBasis* which is derived from *FuncBasis*. *BBasis* augments the services of *FuncBasis*. On the other hand, both *BBasis* and *FuncBasis* have the member function *evaluate(double u)*, however these two functions have different definitions. *BBasis* modifies the *evaluate* function service inherited from *FuncBasis*.

Objects that come from the same class hierarchy are permitted to act as members of a common family, whenever this is appropriate. However, *C++* requires that the declaration of a member function be *virtual*, if it will be redefined further along in a hierarchy. This redefinition of functions is called **polymorphism**.

Sometimes it is desirable to omit a definition for a virtual function in a base class. This can be done by putting `=0` after the member's declaration in place of code. Such a function is called a *pure virtual* function. If a class contains a pure virtual function, it is known as a *virtual class*. No objects of this class type may be declared since the class contains a gap in its definition.

A derived class may provide only some of the definitions for the pure virtual functions that it inherits. It might omit others which would be inappropriate for it to provide. Only when all pure virtual functions have been defined through a chain of class derivations does it become possible to instantiate objects of a particular class.

Overloading is another basic concept of object-oriented programming. It provides efficiency and compactness of syntax by allowing function names to be reused within a class. The various commonly named functions can be distinguished by the type and

number of parameters passed to them. There is no need to invent different names to apply to variants of a single concept. In *C++*, both functions and standard operators can be given overloaded definitions, provided that any ambiguity can be resolved in the context of usage. For example, *evaluate* in the class *FuncBasis* has overloaded definitions which are dependent upon the type of operations.

Recently, **genericity** has become a popular term in object-oriented programming. This term was coined in [Meyer88] to describe a controlled form of macro which is useful in defining collections of similar classes or functions. Because *C++* requires all variables and constants to be declared with a type, it can be inconvenient to construct classes of a generic nature. *Templates* provide a way of getting around this inconvenience. For example, a class can be defined as an array of integer numbers or an array of real numbers. A class or function definition can be headed by a template list of formal tokens, which are then used as type declarations in the definition. Thus a generic class for such array can be created by:

```
template<class DATATYPE>
class array {
    ...
    DATATYPE data;
    ...
}
```

If an instance *array<int> x* is declared, *x* will be an array of integers; if an instance *array<double> x* is declared, *x* will be an array of double precision floating point numbers.

WaSP includes a rich collection of spline types in a well-organized collection of hierarchies. Spline bases are defined separately from classes that assemble splines from a basis and coefficients.

For extensibility, the *C++* spline classes were designed to provide several levels of inheritance so that new classes can be inserted at an appropriate level, and so that programs can use as general a class as possible.

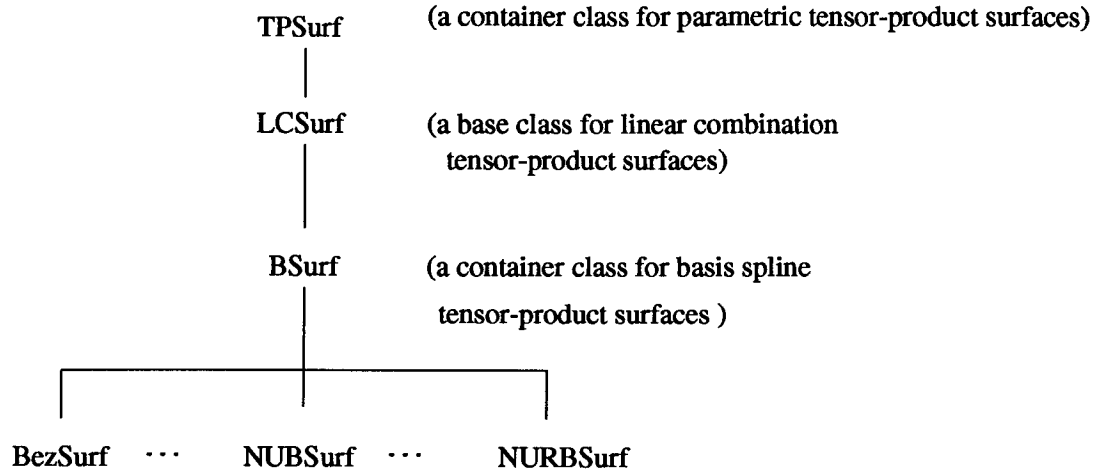


Figure 2.5: The hierarchy for parametric tensor-product surfaces

The goal of reusability led to the separation of the class for a basis from the class for parametric splines. The particular basis used to build spline functions, curves and surfaces is separated from the algorithms and data that a basis needs.

The class *FuncBasis* is the general base class for any function basis. It would be appropriate to derive a class for trigonometric functions from *FuncBasis*. The *BBasis* class is the base class for any basis spline. A basis spline is defined by its degree, its knot vector, and its evaluation algorithm. The evaluation algorithm may use subsidiary information (e.g., a control vertex mesh).

A version of the parametric tensor-product surface hierarchy from WaSP is shown in Figure 2.5. This set of classes is used in our implementation.

The class *TPSurf* is a container class for parametric tensor-product surfaces, from which any other special parametric tensor-product surface can be derived. The class *LCSurf* is a base class for linear combination tensor-product surfaces, derived from the class *TPSurf*. The class *BSurf* is a container class for basis spline surfaces, derived from the class *LCSurf*. It contains a control vertex mesh, *TupleMatrix*, and two linear combination bases in order to describe a completed basis spline surface. A pictorial

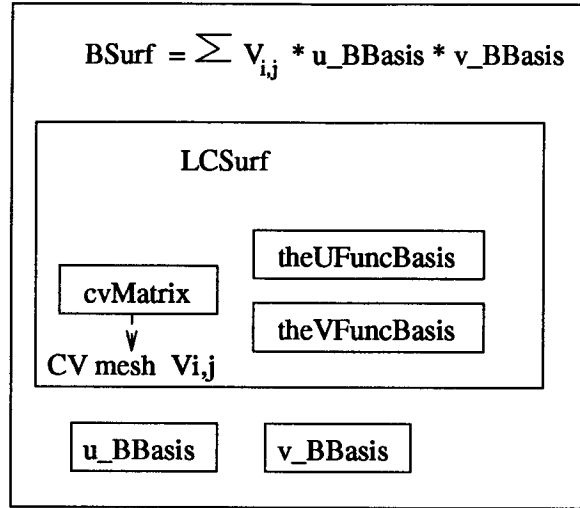


Figure 2.6: The basis spline surface object

representation of a *BSurf* object is shown in Figure 2.6.

The class *TupleMatrix* is a matrix of vectors, which is used to describe the control vertex mesh in a tensor-product B-spline surface. The *TupleMatrix* class is also defined in WaSP.

More information about the above is found in [Bartels91].

Chapter 3

Design and Implementation

This chapter presents the design and implementation for two different quadtree data structures. Both data structures are specifically devoted to representing hierarchical B-spline surfaces.

3.1 Analysis of Hierarchical Surface Representations

This section describes the functional requirements of any implementation of a hierarchical surface, and proposes appropriate data structures for its representation.

The prototypical hierarchical surface editor of [Forsey88] is limited in several ways (see Chapter 1). For example, only a bi-cubic B-spline hierarchical surface can be instantiated and the process of adding new offset methods is not structured. This implementation, however, has served as the model for the functional requirements of a hierarchical surface. The operations can be divided into three groups: surface definition dealing with instantiation, creation and evaluation; surface navigation dealing with accessing CV's; and control node operation dealing with CV attributes and storage.

1. Surface definition:

- create a new hierarchical surface with a given dimension, order, and CV mesh.
- evaluate (position/derivatives) the surface at a parametric point (u, v) .
- traverse one level overlay with a given function.

- traverse an entire hierarchy with a given function.
- generate an overlay (i.e., refine a surface locally around a given CV).
- increase/decrease the dimension (i.e., the number of rows/columns) of a surface

2. Surface navigation:

- get the CV at the position $(i,j,level)^1$.
(this corresponds to the $\vec{V}_{i,j}^{[level]}$ in Section 2.1)
- get the north/south/east/west neighbour of a given CV.
- return neighbours of a given CV within a certain range.
- get the CV at the corresponding parametric location on a different level overlay for a given CV.

3. Control nodes:

- query/set attribute
 - offset vector $\vec{O}_{i,j}^{[k]}$
 - reference vector $\vec{R}_{i,j}^{[k]}$
 - offset method/function \oplus
 - visibility
 - colour
 - movability (i.e., Can the position of the control vertex be altered without causing discontinuities in the surface?)
- get final position (for a given offset method), i.e., $\vec{V}_{i,j}^{[k]}$.

¹ $(i,j,level)$ identifies a CV position in the hierarchy. *level* represents which level overlay this CV lies in; (i,j) represents the indexes in the $m \times n$ matrix of control vertices at that level.

A hierarchical surface is considerably more complicated than a single tensor-product surface. It has:

- multiple levels, each procedurally connected through refinement, offset vector and offset function.
- levels of overlays, each containing one or more continuous spline surfaces.
- an $m \times n$ matrix of CV's for each fully-refined level. If the i^{th} level surface has an $m_i \times n_i$ matrix of CV's, the $(i+1)^{st}$ overlay will have a $(2m_i - u_order + 1) \times (2n_i - v_order + 1)$ matrix of CV's. Here, u_order and v_order are the orders in the u and v parametric directions, respectively. However, depending upon the pattern of refinement, this matrix can be
 - empty,
 - partially filled with an arbitrary clustering of CV's (the minimum size of each cluster is dependent upon the order and type of spline),
 - or full.

Given the wide range of possible distribution of CV's in an overlay, the data structure chosen to represent a hierarchical surface is the quadtree since it has proven very effective in representing rectangular clustering of two dimensional information.

Briefly, a quadtree is a tree whose nodes are either leaves or have at most four children (ordered 1, 2, 3, and 4). If a node in the quadtree is not a leaf, it is called an *internal node*; otherwise, it is called a *leaf node*. A quadtree partitions R^2 into rectangular quadrants recursively according to the similitude of the components within the domain being represented. A general description of quadtrees is provided in Appendix B.

Quadtrees have been used to represent images, point data, areas, curves, and surfaces.

DeFloriani et al. ([DeFloriani82]) discuss a data structure for multilevel surface representation consisting of a nested, triangulated, irregular network ([Lee and Schachter80]) that is used for surface interpolation and can also serve as a mechanism for data compression. Gomez and Guzman ([Gomez79]) use a data structure related to the quadtree that is based upon a recursive subdivision of the surface into four triangles of unequal size. This data structure uses a process that stops when a triangle matches the surface to within a prespecified error. Carlson ([Carlson82]) describes a quadtree-based data structure used to model three-dimensional objects in computer graphics, which allows for the intersection of polygonal objects or other primitive shapes to create more complex objects. Wilhelms ([Wilhelms90]) utilizes octrees, the 3D equivalent of the quadtree, for faster isosurface generation by storing summary information to prevent useless explorations of regions of little or no interest within a volume.

The various kinds of quadtree differ in:

- the type of data represented,
- the decomposition process,
- the resolution (variable or constant).

The decomposition of the domain proceeds either by partitioning into four equal parts (*regular decomposition*) or into unequal parts governed by the input (*irregular decomposition*). For a hierarchical surface, regular decomposition is adopted because of the regular subdivision of the surface by midpoint refinement. Irregular decomposition may prove useful in a hierarchical surface with non-uniform surface refinement, but this is left as a subject for future work.

Recursive local refinement in hierarchical surfaces results in a surface composed of spline patches with different parametric spacing. In a hierarchical surface, each overlay may be only a subset of the full $m \times n$ array of CV's possible for that particular overlay

(i.e., some entries in the array will be null or undefined). A homogeneous region in the array is one where either none or all of the CV's are defined and it is used for determining the regions in a quadtree decomposition.

We consider four different approaches for using quadtree data structures to represent a hierarchical surface. Each approach is characterized according to the type of data stored in each node or decomposition principle (patch definition versus CV definition), and the mechanism used to represent multiple level overlays (one quadtree per overlay versus multiple overlays per quadtree).

If quadtree decomposition is governed by the distribution of patches in each hierarchical overlay, each node in the quadtree represents one or more tensor-product B-spline patch(es). Specifically, each leaf node in the quadtree contains one $s \times t$ control vertex matrix² defining one or more B-spline patch(es). Thus, adjacent nodes in the quadtree will have multiple references to any common CV's.

If quadtree decomposition is based upon a partition of control vertices in a hierarchical surface, each node in a quadtree stores an $s \times t$ control vertex mesh defining part or all of an overlay. This method's advantage is that no control vertex is shared. However, it may be necessary to access several nodes in a quadtree to get one patch definition in an overlay.

To represent the multiple overlays in a hierarchical surface, each level overlay can be represented by a separate quadtree, or the entire hierarchical surface can be stored in a single quadtree, that is, each overlay can be stored in nodes at the corresponding depth in the quadtree. A quadtree representing only one level overlay is called a *single-overlay quadtree*, and one representing the entire hierarchical surface is called a *multiple-overlay quadtree*.

In a *multiple-overlay quadtree*, nodes at a given level represent the corresponding level

²Here $s, t \geq 4$ in the bi-cubic B-spline surface case.

Decomposition principle method of representation	control vertex domain	patch domain
single-overlay quadtree	implemented	*
multiple-overlay quadtree	*	implemented

Table 3.1: Four kinds of quadtree data structures for hierarchical overlays

overlay in a hierarchical surface. Every node, represented by a control node matrix and five pointer fields, is associated with a certain part of the surface. The root node is associated with the root level overlay of the hierarchical surface.

In a *single-overlay quadtree*, each overlay is represented by an independent quadtree that describes the subset of the overlay that has so far been defined by refinement of the parent overlay. Each leaf node in a quadtree representing a given *level* overlay describes the subset of the $2^{2*level}/2^{2*depth}$ possible patches in that level. Here, *depth* is defined to be the depth of the leaf node in the quadtree. All internal nodes are phantom nodes. The entire hierarchical surface structure requires as many quadtrees as there are overlay levels.

In total, there are four kinds of quadtree data structures (Table 3.1) for representing hierarchical overlays, but only two of them have been implemented.

The remainder of this chapter describes how these two approaches are used to represent the same hierarchical surface. Figure 3.7 illustrates the *multiple-overlay quadtree* representation and Figure 3.8 shows the *single-overlay quadtree* representation.

In the following, the detailed operations for forming these two quadtree structures are explored.

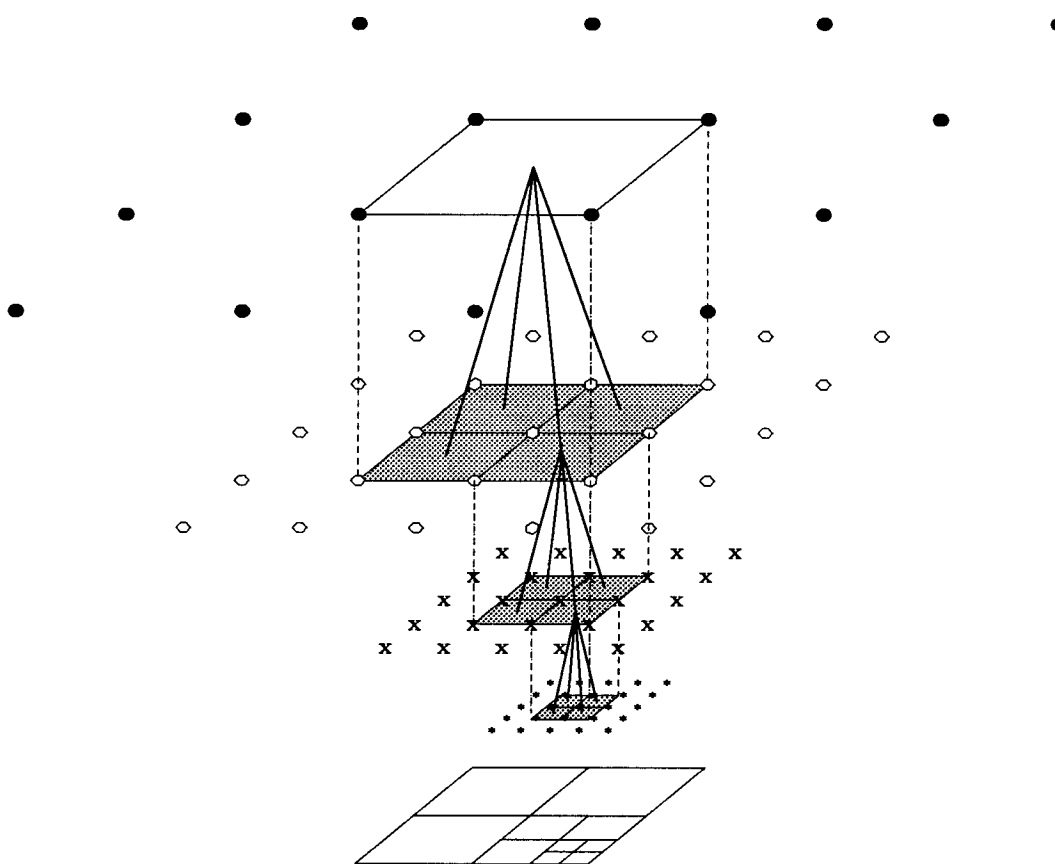
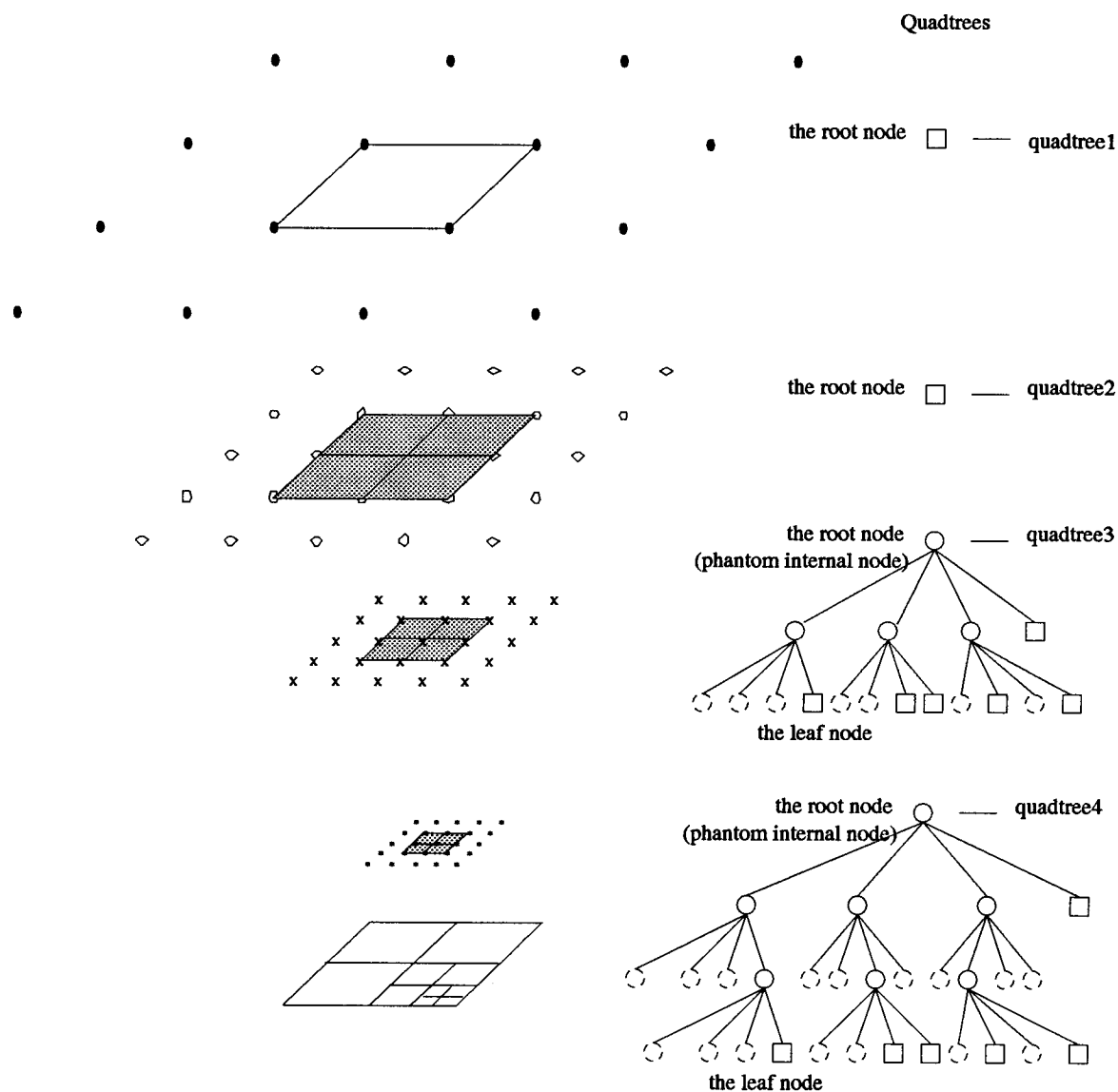


Figure 3.7: A hierarchical surface and its *multiple-overlay quadtree* representation

Figure 3.8: A hierarchical surface and its *single-overlay quadtree* representation

3.2 Overview of Functional Relationships and Structures in the Design

Software support for hierarchical uniform B-spline patches with local refinement restricted to midpoint subdivision has been implemented. The system is written in *C++* and takes advantage of the object-oriented nature of the language. As described in the previous chapter, *C++* allows for the creation of structures that inherit functionality from other structures. The first task in our object-oriented design was to decompose the problem domain into a set of classes.

The criterion of extensibility shaped the various class hierarchies to provide several levels of inheritance. This is arranged to enable new classes to be inserted at an appropriate level (for example, the creation of the class *BSurfQuadTree* derived from the class *QuadTree*) such that programs can use as general a class as possible.

The criterion of reusability led to the decision to separate the class for surface representations (quadtrees) from the class for hierarchical overlays. Because one might want to add new surface representation methods into our hierarchical surface modeller, this design allows for a maximal amount of code reuse.

The criterion of localization shaped decisions about where in the hierarchy the classes should be included and what mechanisms in *C++* should be used to achieve certain goals. The following section examines functional relationships between classes in our design and data structures, and provides one simple example to explain how they fit together.

The organization of our hierarchical overlays provides a mechanism for the creation and manipulation of hierarchical surfaces using Bezier splines, uniform non-rational B-splines and uniform rational B-splines. The following example describes how to construct and represent the particular hierarchical bi-cubic B-spline surface shown in Figure 3.9. This surface has 3 overlay levels obtained in 3 refinement steps around $V_{1,1}^{[0]}$, $V_{1,1}^{[1]}$ and $V_{2,2}^{[2]}$.

Multiple-overlay quadtree representation

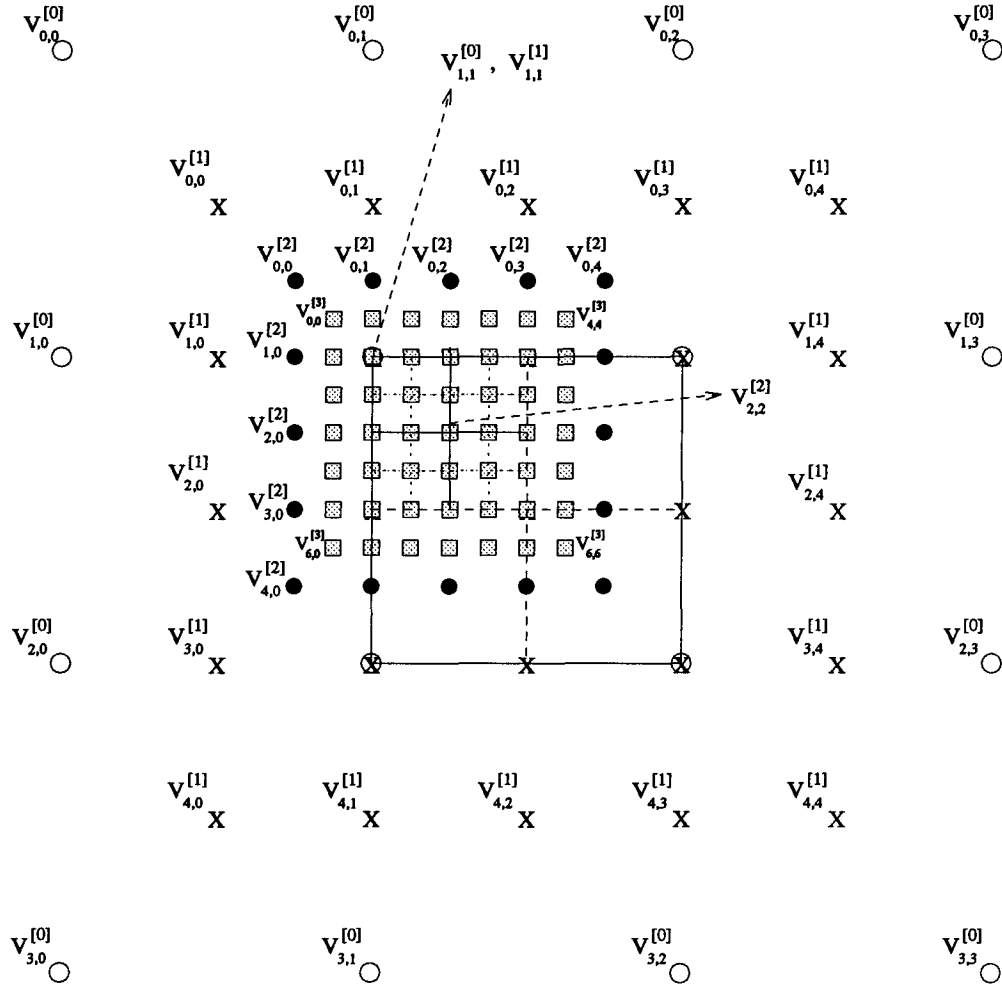


Figure 3.9: A hierarchical bi-cubic B-spline surface

A *multiple-overlay quadtree* hierarchical surface (an instance of the class *HierBOverlay*) begins with one overlay level representing a bi-cubic B-spline patch (surface) defined by a 4×4 control vertex mesh, and is represented by an instance of *multiple-overlay quadtree* (an instance of the class *BSurfQuadTree*). Each overlay at a given level of the surface hierarchy is represented by the nodes of the quadtree at the corresponding depth. Initially, for a new surface (consisting of a single patch), this *multiple-overlay quadtree* has just one root node storing the original patch definition, that is, one 4×4 control node matrix and its corresponding B-spline surface definition. Of course, each node in the B-spline surface quadtree also contains general information found in any quadtree, such as its parent pointer, its four child pointers, the index in its parent, and so on.

Before refinement, the original bi-cubic B-spline patch is represented internally in the *multiple-overlay quadtree*, as shown in Figure 3.10.

To get the hierarchical surface representation shown in Figure 3.9, the region around control node $V_{1,1}^{[0]}$ is refined to create the first overlay consisting of 4 spline patches. The modified data structure in the *multiple-overlay quadtree* is displayed in Figure 3.11 (augmenting the data appearing in Figure 3.10).

In a similar way, the control node $V_{1,1}^{[1]}$ is chosen and refined to get the second level overlay which also has 4 patches. The corresponding *multiple-overlay quadtree* representing this hierarchical surface is shown in Figure 3.12.

The final step in obtaining the hierarchical surface as shown in Figure 3.9, is local refinement around the control node $V_{2,2}^{[2]}$. This refinement produces the desired hierarchical surface represented by the *multiple-overlay quadtree* (Figure 3.13).

Single-overlay quadtree representation

In the *single-overlay quadtree* representation for the hierarchical surface (Figure 3.9), a separate quadtree is required for each overlay level.

A hierarchical surface (an instance of the class *HierBOverlay*) using the *single-overlay*

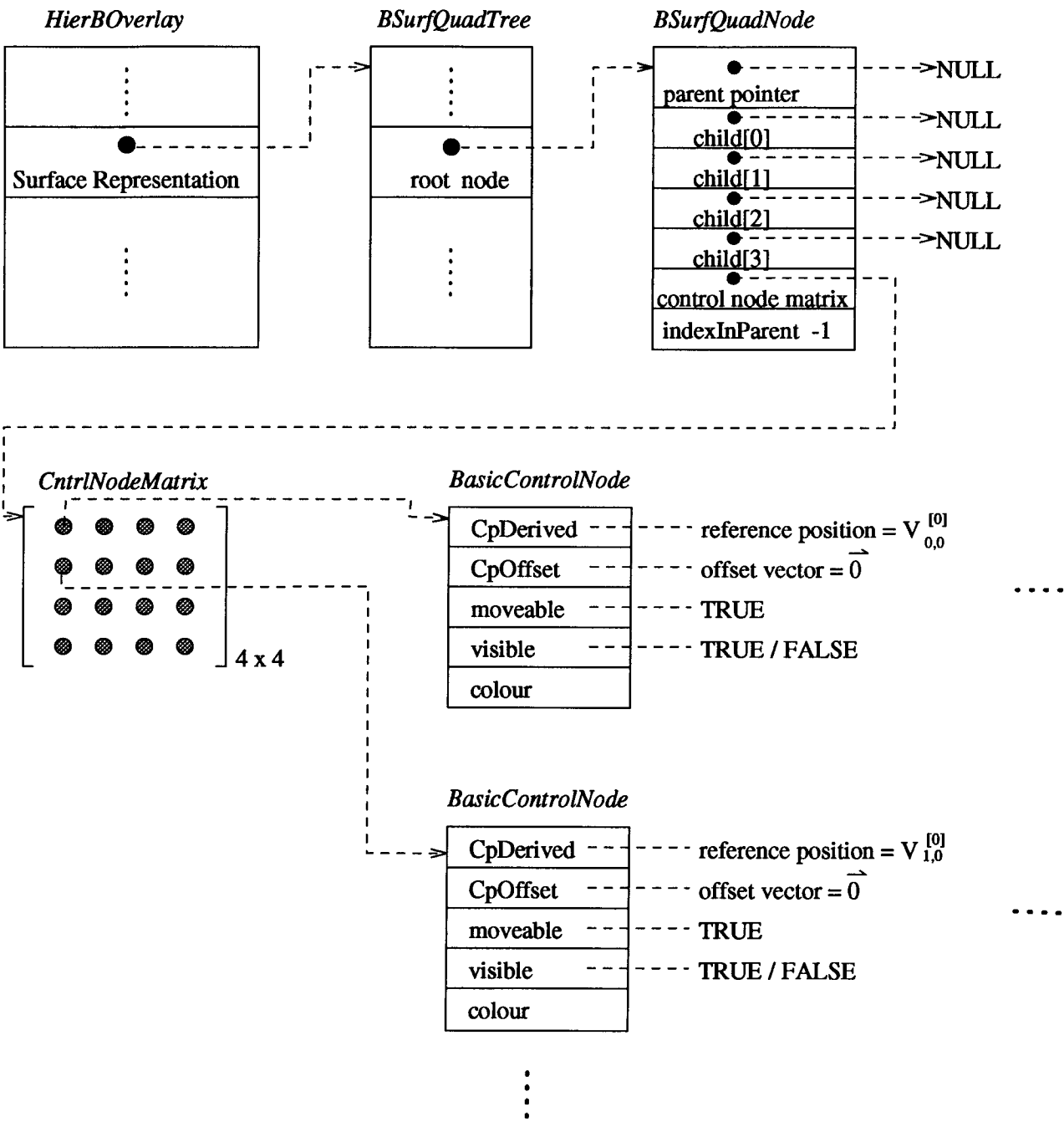
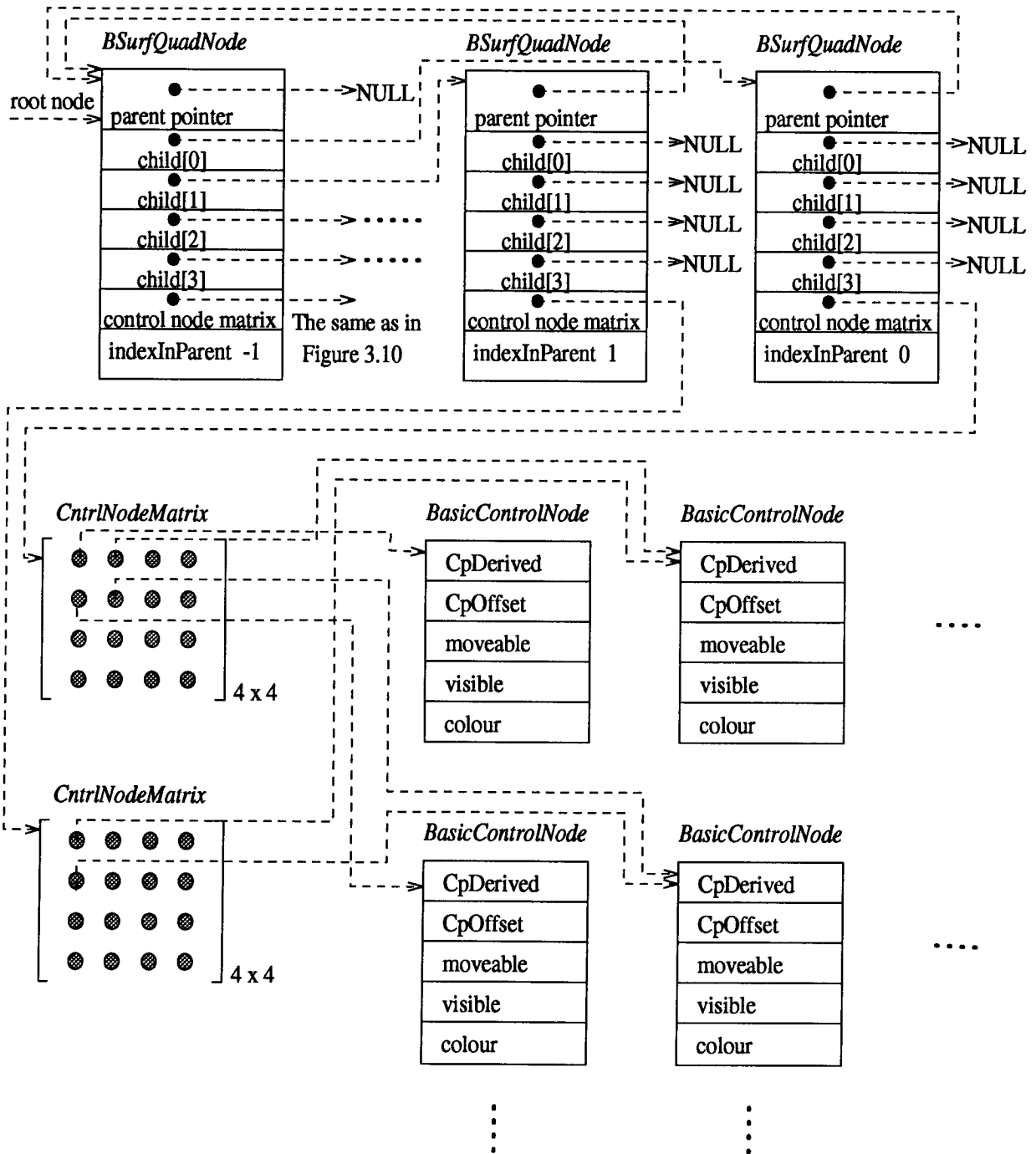


Figure 3.10: The initial hierarchical surface in a *multiple-overlay quadtree*

Figure 3.11: The hierarchical surface in a *multiple-overlay quadtree* after one refinement

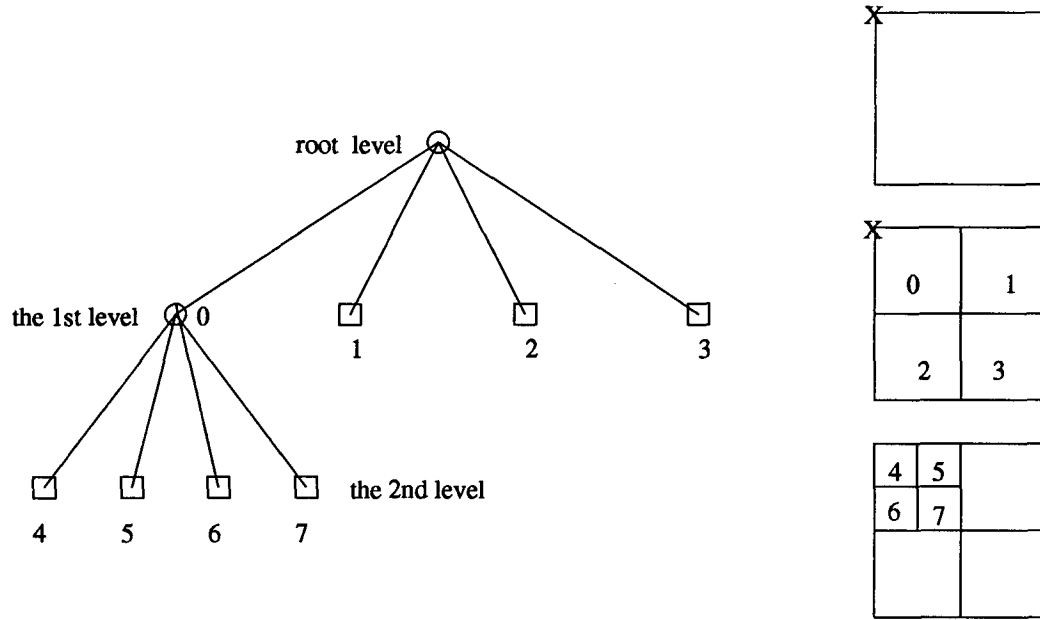
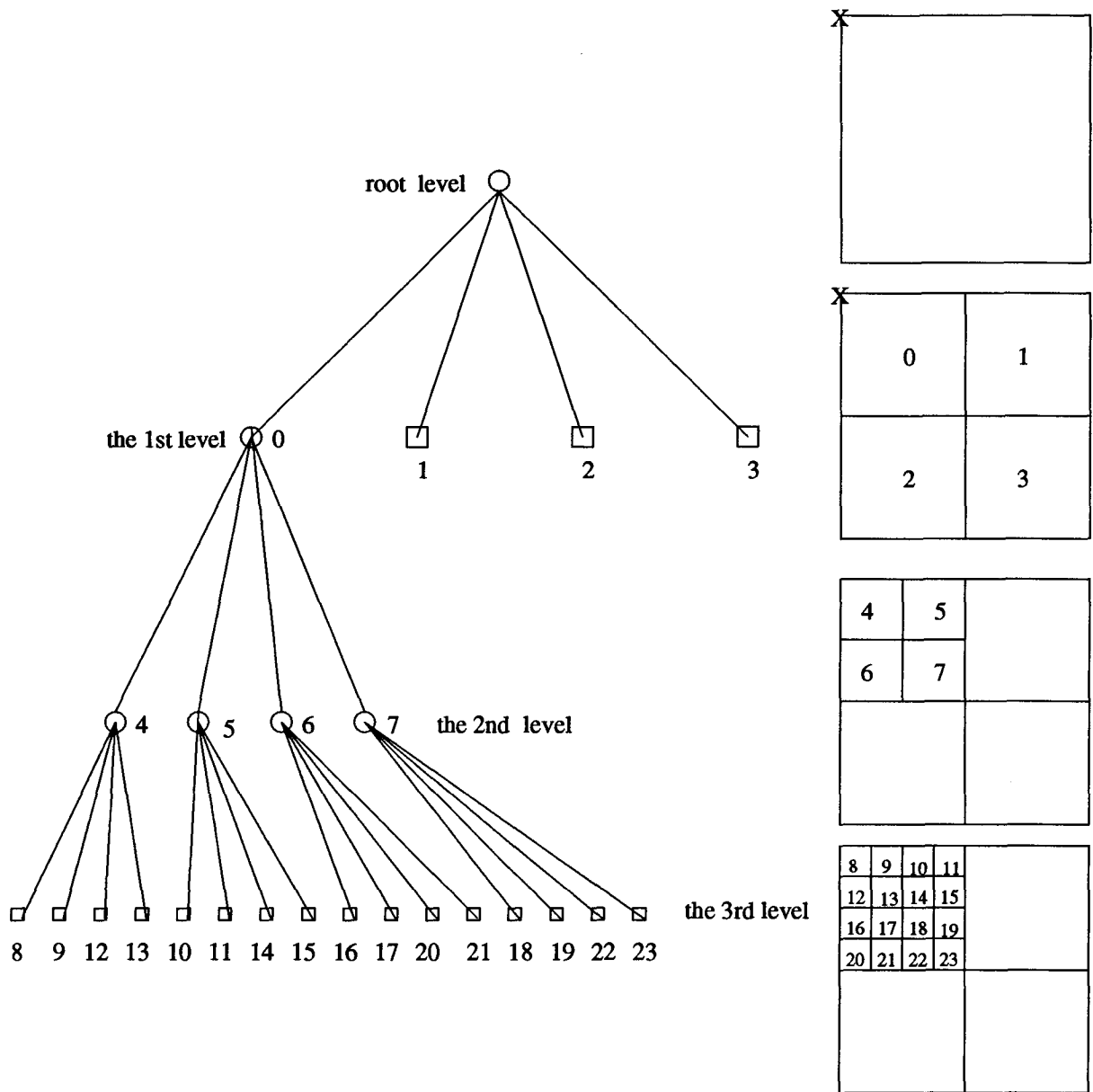


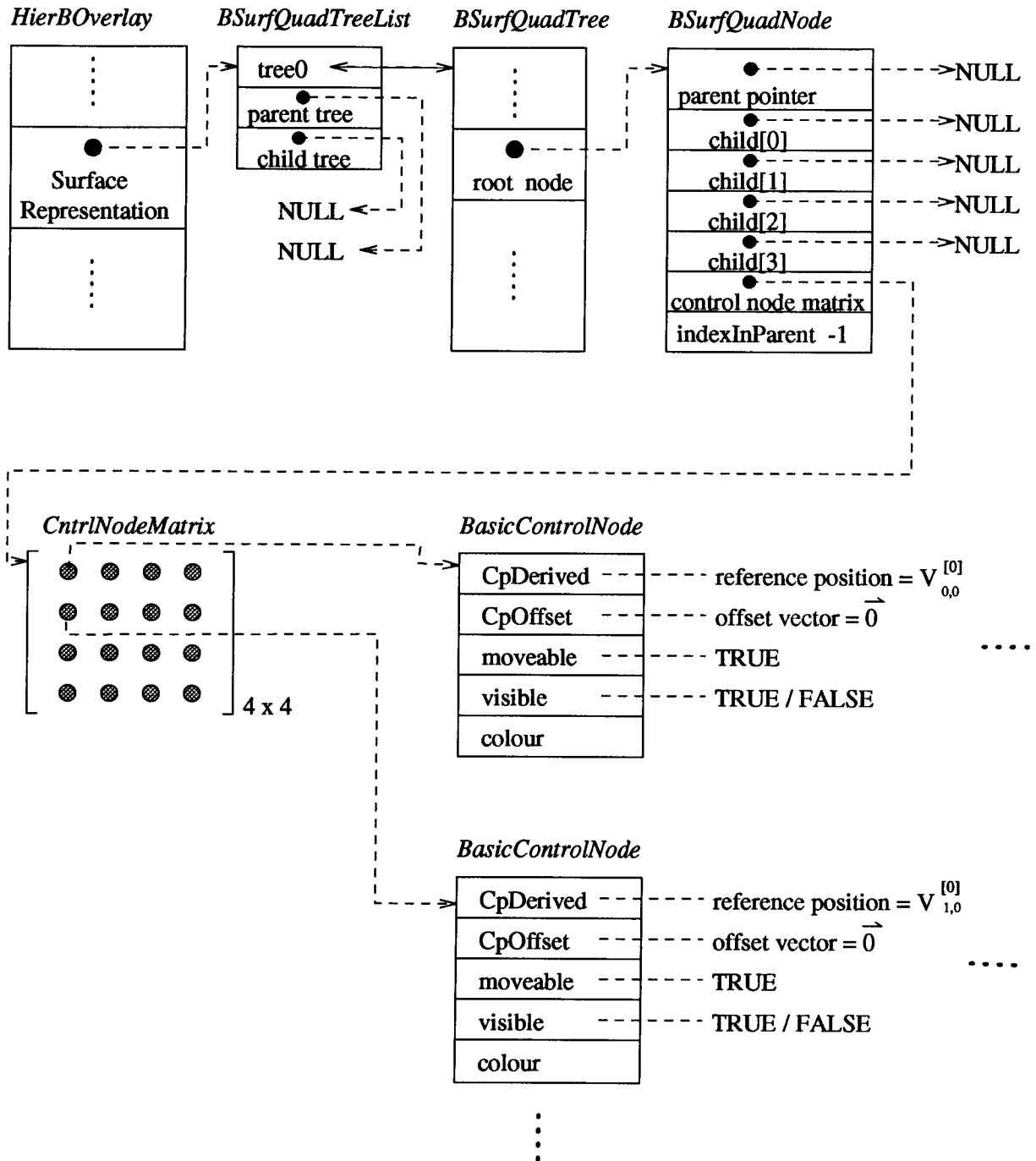
Figure 3.12: The hierarchical surface in a *multiple-overlay quadtree* after the second refinement

quadtree data structure is constructed. This surface (containing only one level of overlay) is stored in one *single-overlay quadtree* list (an instance of the class *BSurfQuadTreeList*) to represent the entire hierarchical surface. For an initial surface consisting of one level overlay, the *single-overlay quadtree* list has only one element, a *single-overlay quadtree* that stores the original surface definition (one control node matrix and its corresponding B-spline surface definition). The *single-overlay quadtree* and *multiple-overlay quadtree* have the same quadtree class definition, except that each node in the *multiple-overlay quadtree* contains one $u_order \times v_order$ control node matrix, whereas a node in the *single-overlay quadtree* could contain up to a $(2^n + u_order - 1) \times (2^n + v_order - 1)$ control node matrix. The remaining objects have the same definition and similar functional relationships as in the *multiple-overlay quadtree*.

Figure 3.14 shows the data structure for a single level single patch surface.

After refining the patches around the control node $V_{1,1}^{[0]}$ to create the first level overlay,

Figure 3.13: The final hierarchical surface in a *multiple-overlay quadtree*

Figure 3.14: The initial hierarchical surface in *single-overlay quadtrees*

a second quadtree is added to the *single-overlay quadtree* list (Figure 3.15).

Refinement around the control node $V_{1,1}^{[1]}$ results in the *single-overlay quadtree* representation shown in Figure 3.16.

Finally, refinement around the control node $V_{2,2}^{[2]}$ results in the hierarchical surface shown in Figure 3.9 and has the corresponding quadtree representation as shown in Figure 3.17 (augmenting the data appearing in Figure 3.16).

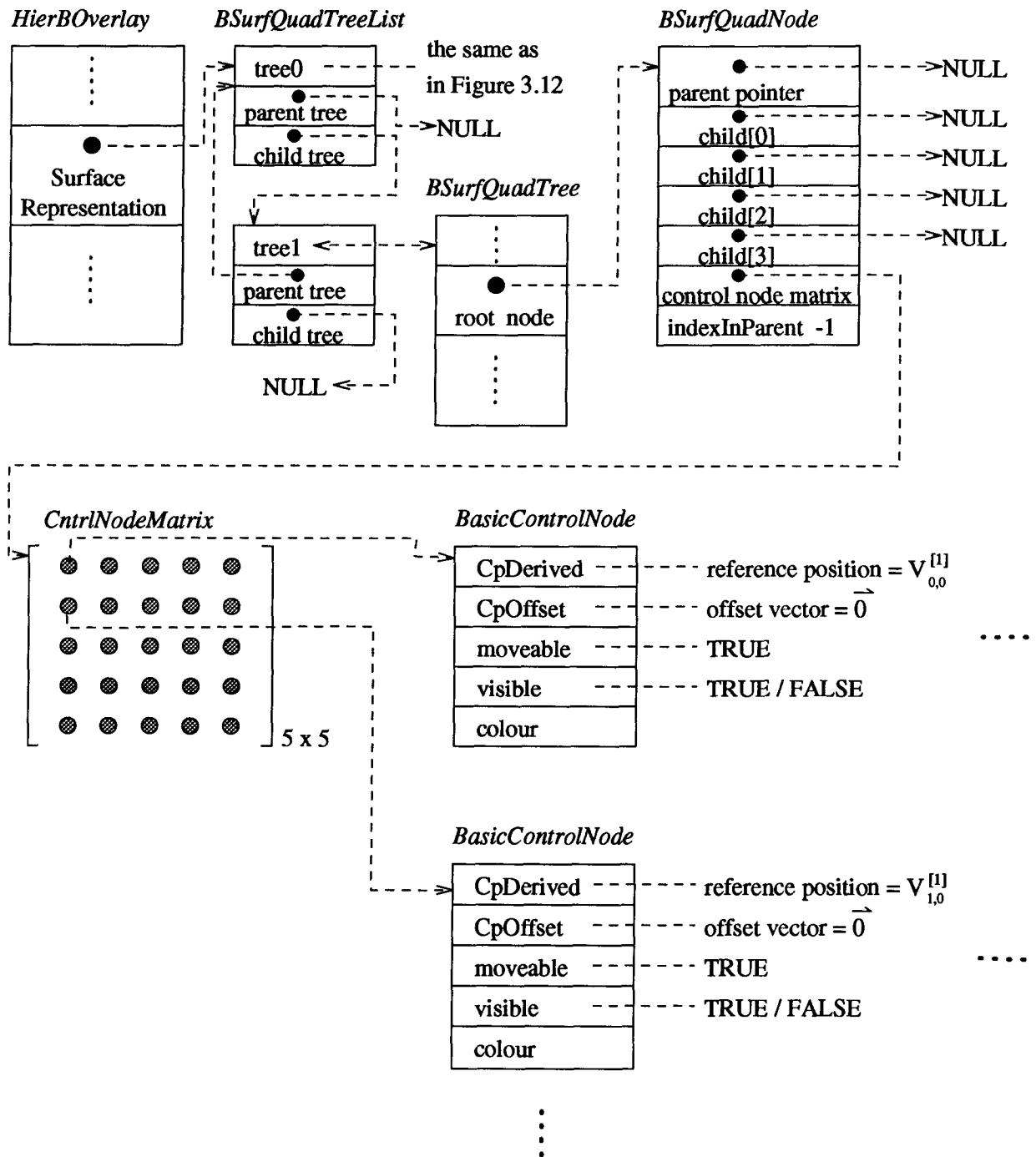
In our design, an object from the class for *hierarchical overlays* holds an object from the class for *quadtree/quadtree list* as member data for surface representation. The quadtree list is represented by a doubly-linked list of quadtrees. In the quadtree data structure itself, there is a pointer pointing to the root node in the tree. Each node in the quadtree has a pointer to its parent and four pointers to its children. Each node contains an object of class *CntrlNodeMatrix* as member data used to define the B-spline patch(es) represented by the node. A pictorial representation of a *HierBOverlay* object in the *multiple-overlay/single-overlay quadtree* structure is shown in Figure 3.18/3.19 .

3.3 Functional Specification for Each Class

This section will explore all the classes related to our hierarchical B-spline surface design and their functional specifications.

3.3.1 Hierarchical Overlay Classes

A simplified version of the hierarchical structure for the class *HierOverlay* (Figure 3.20) shows *HierOverlay*, a general base class for any hierarchical overlay, *HierBOverlay*, a base class for any hierarchical basis spline tensor-product surface, and three leaf classes, *HierNUBOverlay*, *HierNURBOverlay* and *HierBezOverlay*, for hierarchical non-rational B-spline surfaces, hierarchical rational B-spline surfaces and hierarchical Bezier spline

Figure 3.15: The hierarchical surface in *single-overlay quadtrees* after one refinement

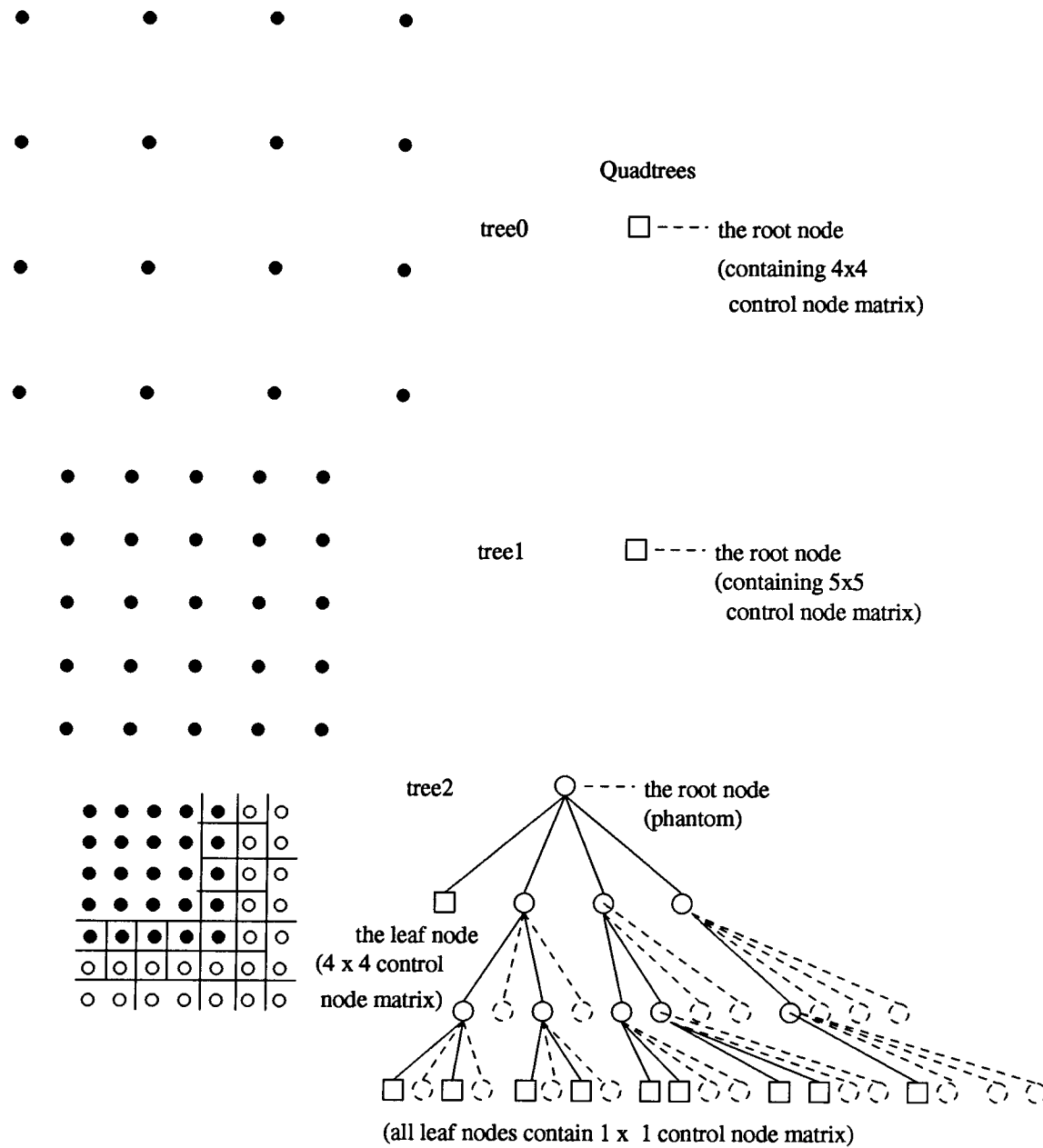
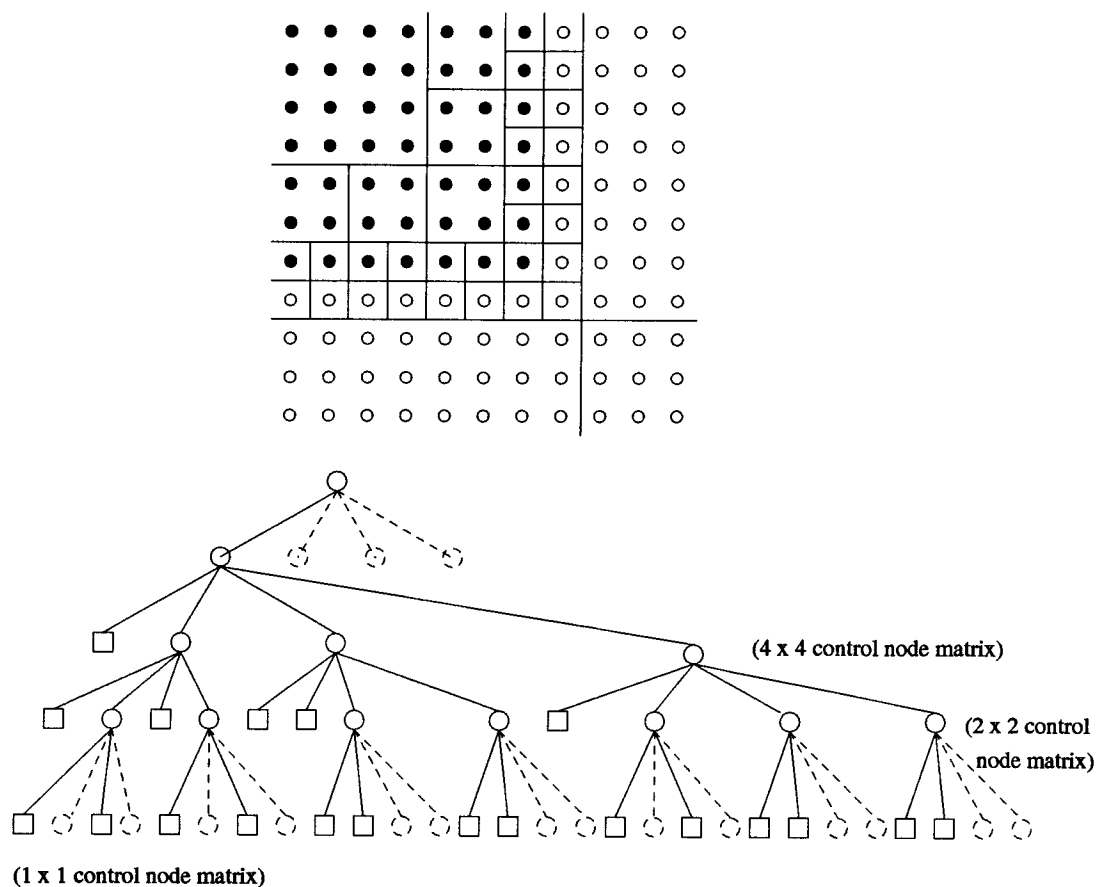
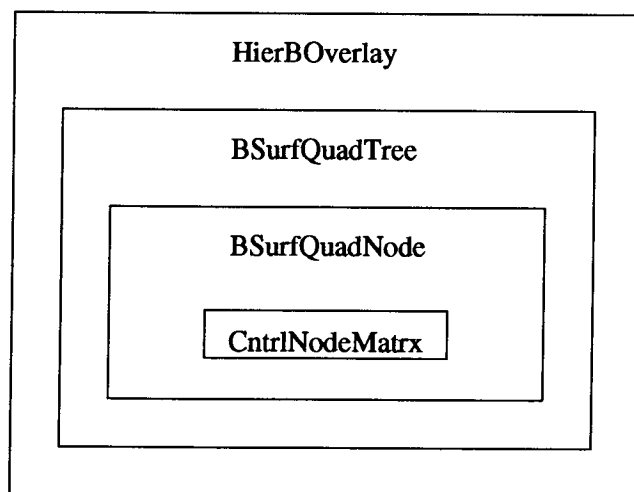


Figure 3.16: The hierarchical surface in *single-overlay quadtrees* after the second refinement

Figure 3.17: The final hierarchical surface in *single-overlay quadtrees*Figure 3.18: A hierarchical B-spline surface object in the *multiple-overlay quadtree*

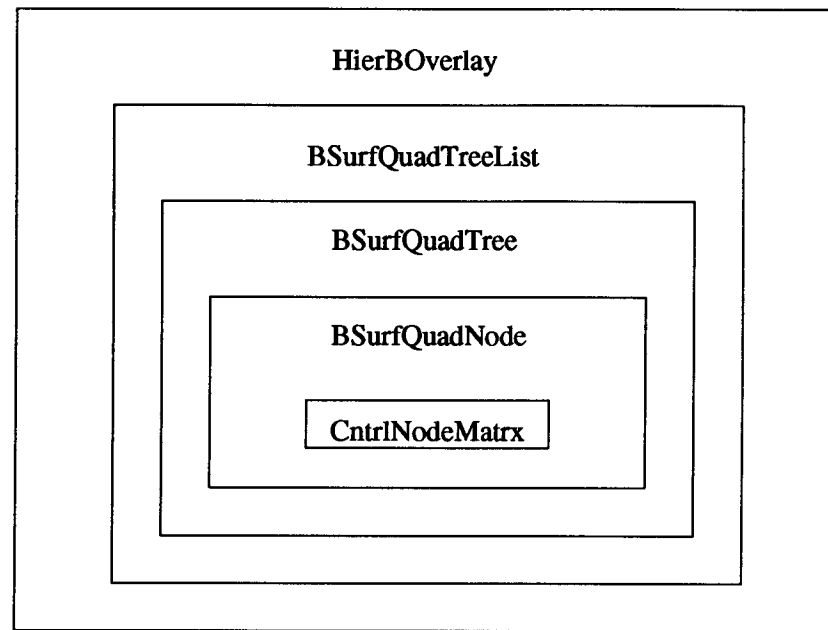
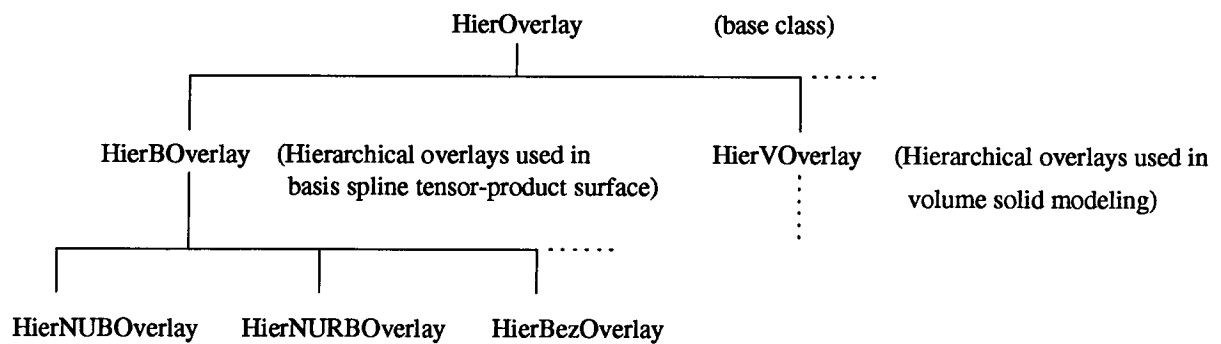
Figure 3.19: A hierarchical B-spline surface object in *single-overlay quadtrees*

Figure 3.20: HierOverlay Class Structure

surfaces, respectively.

The HierOverlay Class

The *HierOverlay* class describes any type of hierarchical overlay. The following code fragment shows key members of this class:

```
class HierOverlay {  
    protected:  
        int name;  
        // Identify this hierarchical overlay -- a unique name.  
    public:  
        virtual short totalLevel()=0;  
        // Return the total number of levels of overlays in this hierarchy.  
        virtual void traversal( HierTravFn function )=0;  
        // Traverse this hierarchy with a given function.  
        inline int getName() const;  
        inline void setName( int i );  
        // Get or set this hierarchical overlay's identification.  
};
```

All the hierarchical overlay structures have some common operations, such as retrieving the total number of overlays in the hierarchy, or traversing the hierarchy with a given function, but employ different methods to perform these operations. Therefore, we make these methods pure virtual functions in this abstract class. This ensures a definition for each method will be provided by one of *HierOverlay*'s derived classes.

The HierBOverlay Class

HierBOverlay is a base class for any hierarchical basis spline surface. In this and all subsequent classes, only a selected subset of the member functions will be shown. Destructors and some variants of other member functions designed for special cases or efficiency do not appear for the sake of brevity.

```

template <class SurfaceRepresentation>

class HierBOverlay : public HierOverlay {

protected:

    SurfaceRepresentation *tree;
    // Pointer to the whole hierarchical data structure.

    RWBoolean symmetry;
    // Indicate whether all the operations are handled symmetrically.

public:

    HierBOverlay( BSurf* );
    HierBOverlay( BSurf * );
    HierBOverlay( const HierBOverlay* );
    // Copy constructor creates an instance pointing to the same surface.

    short totalLevel();
    // Return the total number of all the overlays in this hierarchy.

    void updateCV( short level, int i, int j, DoubleVec vector );
    // Edit a CV at position (level,i,j) in the hierarchical surface.
    // Update its offset vector and all affected control nodes in finer overlays.

    void traversal( TravFn fn );
    // Traverse the hierarchy with a given function.

    void leavesTraversal( TravFn fn );
    // Traverse all leaves of hierarchy applying fn to each leaf node of.
    // the quadtree. fn returns a boolean. If the boolean is TRUE, the
    // traversal continues; if it is false, the traversal terminates.

    void nodesTraversalInLevel( short i, TravFn fn );
    // Traverse all nodes in the level "i" applying fn to each node.

    void refinement( short level, int i, int j );
    // Only refine where needed according to the specified CV.

    CntrlNodeList* CVInDownLevel( short level, int i, int j );
    CntrlNodeList* CVInUpLevel( short level, int i, int j );
    // Get a CV in the down/up level from a given CV(same u,v location).

    ControlNode* CVInDownPatch( short level, int i, int j );
    ControlNode* CVInUpPatch( short level, int i, int j );
    // Get a CV in the down/up patch from a given CV(same patch definition).

    ControlNode* CVNeighbour( short level, int i, int j, int flag );
    // Return north/south/east/west neighbour of a given CV
    // according to flag = NORTH/SOUTH/EAST/WEST.

    CntrlNodeMatrix* CVNeighbours( short level, int i, int j,

```

```

                                int low_i, int high_i,
                                int low_j, int high_j );
// Return given range neighbours of a given CV.

ControlNode* CVNavigate( short level, int i, int j );
// Return the control node at (level,i,j).

DoubleVec getUVParameters( short level, int i, int j );
// Get the corresponding (u,v) values in hierarchy.

virtual void evaluate( double u, double v, Triple * );
virtual void evaluate( double u, double v, int du, int dv, Triple * );
virtual void normal( double u, double v, Triple * );
virtual void curvature( double u, double v, double * );
virtual void curvMaxMin( double u, double v, Triple *, Triple * );
// Evaluate the surface at a given point in parametric space.

inline void setSymmetry( RWBoolean value );
inline RWBoolean getSymmetry();
// Get or set the symmetry attribute in this hierarchy.

inline BSurfQuadTree *refQuadTree() const;
};

```

The most important member function in the *HierBOverlay* class is *refinement*. Only via *refinement* can we get multi-level overlays in the hierarchical surface. Refinement occurs only where it is necessary to gain finer control over the B-spline surface during editing. Another important member function is *updateCV*, which allows us to edit the final position of a CV in the hierarchical surface. The class *HierBOverlay* also provides a number of basic functions including various kinds of traversal of the hierarchy, getting the control node via its position (*level,i,j*), finding the neighbour(s) of a given CV, and evaluating a point in the hierarchical surface corresponding to the point (*u,v*) in its parametric space. The initially input B-spline surface is defined by its degree/order, knots and control vertex mesh.

An object *tree* of the template class *SurfaceRepresentation* provides the entire hierarchical surface representation. In the *multiple-overlay quadtree* representation, the template class *SurfaceRepresentation* is specified as the class *BSurfQuadTree*. In the

single-overlay quadtree representation, the template class *SurfaceRepresentation* is specified as the class *BSurfQuadTreeList*. For example, if we want to declare an object *x* as a hierarchical surface using the *multiple-overlay quadtree* structure, we could specify **HierBOverlay<BSurfQuadTree> x**. Similarly, if we want to declare an object *x* as a hierarchical surface using the *single-overlay quadtree* structure, we need to specify **HierBOverlay<BSurfQuadTreeList> x**. All of these quadtree-related classes are defined in the *QuadTree* class library. The class *BSurf* provides a general B-spline surface description. It is defined in WaSP. The class *ControlNode* and the class *CntrlNodeList* provide the control vertex and the list of control vertices representations, respectively. They are defined in the *SupportClasses* library.

3.3.2 Quadtree Classes

A version of the hierarchy for the class *QuadTree* is shown in Figure 3.21.

The figure shows *SNTree*, a general base class for an N-ary tree, and *BSurfQuadTree*, a class for the quadtree representing a hierarchical B-spline surface in the *multiple-overlay quadtree* representation, or representing one overlay of a hierarchical B-spline surface in the *single-overlay quadtree* representation. Correspondingly, each *BSurfQuadTree* has its own node class definition. The node classes themselves have the same hierarchical structure.

The SNTree Class

The *SNTree* class from WaSP describes an N-ary tree data structure, each node of which has one parent node and any number of child nodes. The key members of this class are:

```
class SNTree {
    protected:
        SNTreeNode *root;
```

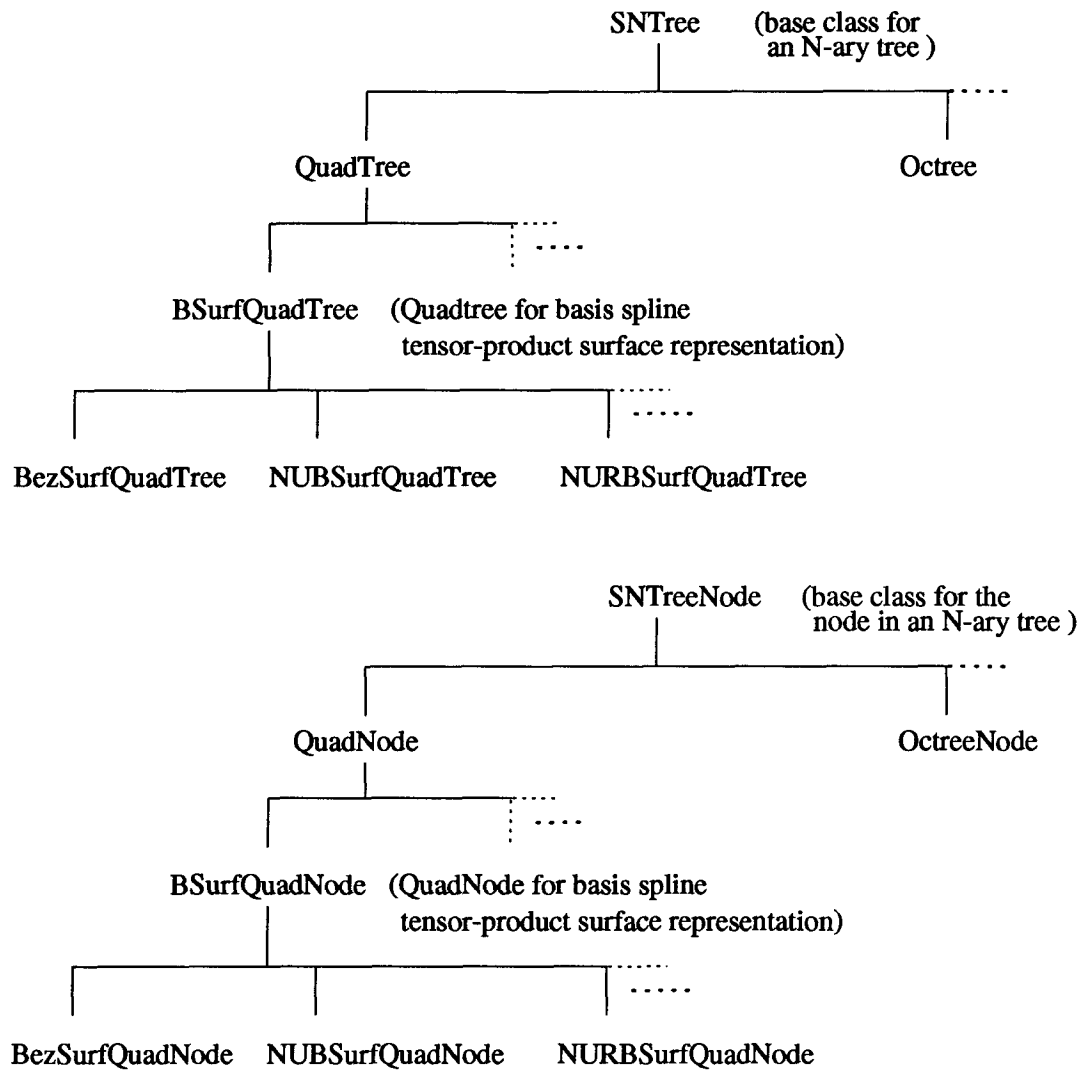


Figure 3.21: QuadTree Class Hierarchy

```
// Pointer to the root of the tree.

public:

    void insertRoot (SNTreeNode& newRoot);
    // Make newRoot the root of this tree.

    RWBoolean insert(SNTreeNode& parent,int childPos,SNTreeNode& element);
    /* Insert an element as the childPos child of parent. All current
       children of element are discarded.
       Insert returns TRUE if successful, FALSE if something went wrong. */

    RWBoolean removeSubTree (SNTreeNode& node, SNTreeTravFn fn = NULL);
    /* Remove node from the tree. All of node's children are removed too.
       returns TRUE if successful, FALSE otherwise
       Each node in the subtree is removed from the tree recursively by
       doing a prefix traversal on the subtree and calling fn on each node.
       If no function is supplied a default routine is used. */

    SNTreeNode *removeSubTree (SNTreeNode& parent, int childPos,
                               SNTreeTravFn fn = NULL);
    /* Remove the childPos child of parent. All of the removed node's
       children are removed as well. Returns a pointer to the removed
       root of the subtree or NULL if something went wrong
       Each node in the subtree is removed from the tree recursively by
       doing a prefix traversal on the subtree and calling fn on each node.
       If no function is supplied a default routine is used. */

    SNTreeNode *remove (SNTreeNode& parent, int childPos);
    /* Remove the childPos child of parent. All of the removed node's
       children remain in the tree and become children of node's parent.
       Returns a pointer to the removed node/NULL if something went wrong. */

    RWBoolean remove (SNTreeNode& node);
    /* Remove node from the tree. All of node's children remain
       in the tree and become children of node's parent.
       Returns TRUE if successful, FALSE otherwise. */

    RWBoolean moveSubTree(SNTreeNode& node,SNTreeNode& destParent,int pos);
    /* Move the subtree rooted at node to be the pos-th child of parent.
       For the operation to be successful, parent must not already have a
       child in that position. Returns TRUE if successful, FALSE otherwise. */

    RWBoolean isEmpty ();
    // Return TRUE if the tree is empty.

    RWBoolean isNodeInTree (SNTreeNode& node);
    // Check to see if the node is really in the tree.

    int whichChild (SNTreeNode& parent, SNTreeNode& node);
    /* Search for the child number of node in parent.
       If node isn't a child of parent then return -1. */
```

```

void prefixTraversal (SNTreeTravFn fn);
void postfixTraversal (SNTreeTravFn fn);
/* Traverse the tree in a pre or postfix order applying fn to each
   element of the tree. fn returns a boolean. If the boolean is TRUE,
   traversal continues, if it is FALSE, traversal terminates. */
};

```

SNTree provides many basic operations for a tree data structure, such as removal of a node, insertion of a node, subtree movement, traversal in pre/postfix order, and so on. Since these operations are needed in all the tree data structures, we could reuse a large amount of code if we derived the *QuadTree* and *Octree* classes from *SNTree*.

The *SNTreeNode* class from WaSP defines a node in the N-ary tree class *SNTree*. It can be inserted into trees of the type *SNTree*. The key members of the class *SNTreeNode* are:

```

class SNTreeNode {
protected:
    int numChildren;

    SNTreeNode **children;
    // Pointers to the children of this node.

    SNTreeNode *parent;
    // Pointer to the parent of this node.

public:
    SNTreeNode();
    SNTreeNode(const SNTreeNode& n);
    SNTreeNode(int numchildren);
    // The third constructor constructs a node with the specified number of children.

    RWBoolean hasChildren(void) const;
    // Return true if the node has any children, false otherwise.

    inline SNTreeNode *getParent(void) const;
    // Return the parent of this node.

    virtual SNTreeNode *getChild(int i) const;
    // Return the ith child of this node.

    inline int getNumChildren(void) const;
    // Return the number of children this node can accommodate.

```

```

    int growNumChildren(int newNumChildren);
    // Increase the number of children this node has to newNumChildren.
    // newNumChildren must be larger than the position of the last
    // non-NULL child. Returns old numChildren.
};

```

The class *SNTreeNode* holds basic information needed for a node in a tree, such as the number of children, its parent pointer, and its child pointers. It also provides basic operations for the node, like getting its parent, fetching one of its children, obtaining the number of children, and judging whether or not it has children. It would be an appropriate base class for other tree node classes.

The QuadTree Class

The class *QuadTree* is used to implement a general class of quadtree data structures which are based on the principle of recursive decomposition of space into four partitions.

The key members of the class *QuadTree* are:

```

class QuadTree : public SNTree {
public:
    QuadTree();
    // Constructor for the empty tree.

    QuadTree( const QuadNode& );
    QuadTree( QuadNode * );
    QuadTree( const QuadTree& );
    // For copying.

    inline QuadNode *rootNode() const;
    // Get the root node in the current tree.

    void leavesTraversal( QuadTreeTravFn fn );
    /* Traverse all leaf nodes in the tree applying fn to each leaf node.
       fn returns a boolean. If the boolean=TRUE, the traversal continues;
       if it is false the traversal terminates. */

    void nodesTraversalInLevel( short level, QuadTreeTravFn fn );
    /* Traverse all nodes in the depth "level" of the tree applying fn
       to each node. fn returns a boolean. If the boolean is TRUE,
       the traversal continues; if FALSE, the traversal terminates. */
};

```


Since *QuadTree* inherits all the operations from the class *SNTree*, we do not need to provide additional operations specifically for manipulating quadtrees. In order to make it easy for reading and using, we list *prefixTraversal* and *postfixTraversal* here again with two other traversal operations, *leavesTraversal* and *nodesTraversalInLevel*. Note these methods are simply calling the corresponding functions in the base class *SNTree*. *SNTree* also provides another basic function *rootNode* to let us access the root node of a quadtree.

The key members of the class *QuadNode* are:

```
class QuadNode : public SNTreeNode {
protected:
    int indexInParent;
    // Identify which child of its parent this node belongs to;
    // If it has no parent, indexInParent = -1.

public:
    inline void setParent( QuadNode * );

    void putNewChild( QuadNode *, int );
    // Put the given node as the indexed child. Return this child.

    RWBoolean isLeaf() const;
    // Return true if the node is a leaf.

    inline int index() const;
    // Return which child of its parent this node belongs to.

    inline int depth() const;
    // Return this node's depth in the tree.
};
```

Common function members which are often used in the quadtree are listed in the class *QuadNode*. This ensures they are localized and easy to read and use.

The BSurfQuadTree Class

This class implements the *multiple-overlay quadtree* approach for hierarchical overlays. In the hierarchical surface design, the class *BSurfQuadTree* is derived from the general

class *QuadTree*. The data stored in each node represents some of the B-spline patches. Decomposition is based on subdividing the parametric space, which is reflected in the original surface. Resolution depends on the refinement of a surface in the *multiple-overlay quadtree* representation. The key members of the class *BSurfQuadTree* are:

```
class BSurfQuadTree : public QuadTree {

    protected:

    ** short virtualHeight;
        // Indicate which level contains the actual surface node.
        // In the single-overlay quadtree, we don't need this member.

        short totalHeight;
        // Indicate the total height in this quad tree.

    ** int patchNoInU, patchNoInV;
        // Indicate how many patches are contained in the root level.
        // In the single-overlay quadtree, we don't need these two members.

    public:

        BSurfQuadTree( BSurf& );
        BSurfQuadTree( BSurf * );
        // Set the root node's surface to the given one, constructor

        BSurfQuadTree( const BSurfQuadNode&, BSurfQuadTree *tree );
        // Set the given node to the root node, copy its subtree

    ** short actualHeight();
        // Return the actual height of the tree which represents the
        // different refined overlays.
        // In the single-overlay quadtree, we don't have this function.

        short height();
        // Return the height of the tree.

        BSurfQuadNode *getQuadNode( **short level, int i, int j );
        // Get the quadnode which patch is identified by CV (level,i,j)
        // In the single-overlay quadtree, we don't have the parameter - level.

        ControlNode* CVNavigate(**short level,int i,int j,HierBOverlay *x);
        // According to this control vertices's index (i,j), return its
        // corresponding control node.
        // In the single-overlay quadtree, we don't have the parameter - level.

        void updateCV( **short level, int i, int j, DoubleVec );
        // Set the double vector to a control node offset, given
        // index (i,j). We also need to modify all the affected
        // reference positions in the finer level(s).
```

```

// In the single-overlay quadtree, we don't have the parameter - level.

CntrlNodeMatrix* CVNeighbours( **short level, int i, int j,
                                int low_i, int high_i,
                                int low_j, int high_j );
// According to this control node index (i,j) , return the
// corresponding control node matrix , which index range is from
// (i+low_i, j+low_j) to (i+high_i, j+high_j)
// In the single-overlay quadtree, we don't have the parameter - level.

CntrlNodeList* CVInDownLevel(short level,int i,int j,HierBOverlay *x);
// Return a CV in the down level from a given CV (same u, v location).

CntrlNodeList* CVInUpLevel(short level,int i,int j,HierBOverlay *x);
// Return a CV in the up level from a given CV (same u, v location).
};

```

The class *BSurfQuadTree* uses the member *virtualHeight* from the *multiple-overlay quadtree* structure to handle an input initial surface which is not *regular* (if the number of patches in the *u* or *v* parametric direction is not 2^n , phantom internal quadtree nodes are needed to obtain the desired *regular* decomposition). In the *single-overlay quadtree* structure, this member is not needed since it is identified in the class *BSurfQuadNode* in order to merge different patches during refinement. The members *virtualHeight*, *patchNoInU* and *patchNoInV* are used to identify phantom internal nodes in the *multiple-overlay quadtree* representation. All other member functions in this class support the corresponding operations in the class *HierBOverlay*, no matter what kind of quadtree representation is adopted.

The key members of the class *BSurfQuadNode* are:

```

class BSurfQuadNode : public QuadNode {
protected:
    CntrlNodeMatrix *CVnodes;
    // Store the control nodes' information related to this surface patch.

public:
    BasicControlNode *getInternalControlNode( int i, int j );
    // Get the corresponding internal control node in the quadtree.

```

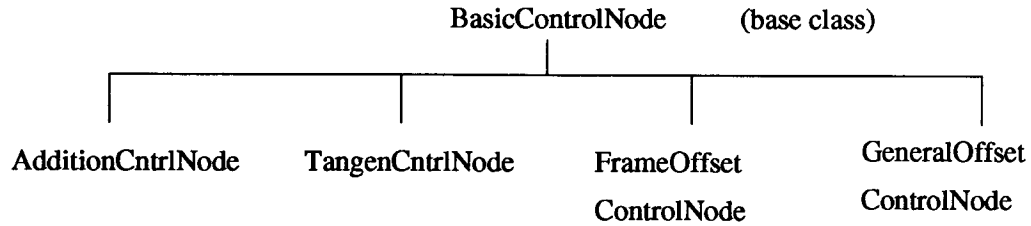


Figure 3.22: The hierarchy for a control node definition

```

CntrlNodeMatrix* getCntrlNodeMatrix();
// Return the related control node matrix.

TupleMatrix getCpFinalMat();
// Get the current CV matrix which patch is defined in this quad node.
};

```

Each node in the quadtree for representing hierarchical B-spline surfaces describes a B-spline patch(es). In other words, the class *BSurfQuadNode* contains the member *CVnodes*, an object of the class *CntrlNodeMatrix*. The other function members provide operations for surface representation and offset-referencing method. In the *single-overlay quadtree*, each node holds two members *patchNoInU* and *patchNoInV* to store the actual number of patches in the *u* or *v* parametric direction, since they might not be 2^n .

3.3.3 The Support Classes

To take advantage of the offset-referencing mechanism and gain better offset structure in our hierarchical surface design, we define classes suitable for describing a control node in a hierarchical surface. This definition supports various kinds of offset methods, such as vector addition, tangent plane, skeletal frame and dynamic function. A version of the hierarchy for a control node class is shown in Figure 3.22.

The key members of the base class *BasicControlNode* are:

```

class BasicControlNode {
private:

```

```

    void setCpDerived( DoubleVec vector );
    // Set the control node's derived position.

protected:

    DoubleVec CpDerived;
    // Obtained by subdivision from parent.

    DoubleVec *CpOffset;
    // Set by editing of this control node.

    RWBoolean moveable;
    RWBoolean visible;
    long color;
    // Indicate the control node's attributes.

public:

    BasicControlNode( unsigned dimension=3 );
    BasicControlNode( DoubleVec vector );
    // If we are given the derived vector.

    BasicControlNode( const BasicControlNode& );
    // Constructor.

    inline DoubleVec getCpDerived() const;
    // Get the control node's derived position.

    inline DoubleVec getCpOffset();
    void setCpOffset( DoubleVec vec );
    // Get or set the control node's offset vector.

    BasicControlNode& operator=( const BasicControlNode& x );
    // Set "x" to the current BasicControlNode.

    BasicControlNode copy() const;
    // Make a copy for the current control node.

    RWBoolean getBooleanFlag( CntrlNodeFlag flag );
    void setBooleanFlag( CntrlNodeFlag flag, RWBoolean value );
    // Note the definition : enum CntrlNodeFlag {VISIBLE, MOVEABLE}.

    long getAttribute( CntrlNodeAttr attr );
    void setAttribute( CntrlNodeAttr attr, long value );
    // Note the definition : enum CntrlNodeAttr {COLOR}.

    virtual DoubleVec getCpFinal();
    // Use the offset method to calculate the final CV position.
};

```

The most important member function in the class *BasicControlNode* is *getCpFinal*. This function calculates the control node coordinates from $\vec{R}_{i,j}$, $\vec{O}_{i,j}$ and \oplus .

The class *BasicControlNode* is the type for the private control node. It contains the following information:

- reference position (obtained by subdivision from the parent patch),
- offset vector pointer,
- how to get a reference position,
- how to get/reset an offset vector,
- how to get/set an attribute (attr_name,value),
- how to calculate a final position (apply offset method to offset).

The class *ControlNode* is used as the type for the public control node in the application, and contains more information that is not in a *BasicControlNode*:

- a pointer to the hierarchical surface that contains the node,
- which level overlay of the hierarchy contains the node,
- the node's position (*i,j*) in the current level,
- an internal/private control node object.

The key members of the class *ControlNode* are:

```
class ControlNode {
    private:
        int ith, jth;
        // Represent the control node's position in one level surface overlay.

        short level;
        // Represent which level of the hierarchy the control node lies in.

        HierBOverlay *surface;
        // Identify which hierarchic surface this control node lies in.

        BasicControlNode *node;
        // The basic control node information responds to this Control Node.
```

```

public:

    ControlNode copy() const;
    // Copy the current control node.

    ControlNode operator=( const ControlNode& x );
    // Should point to the same node as BasicControlNode.

    inline HierBOverlay *getSurface() const;
    // Get which hierarchical surface this control node lies in.

    IntVec getPosInHier();
    // Get the control node's position (i,j,level) in the hierarchy.

    inline short whichLevel() const;
    // Get which level of the hierarchy this control node lies in.

    inline BasicControlNode *getControlNode() const;
    // Get the internal control node.

    DoubleVec getCpFinal();
    // Get this control node's final position in world coordinate frame.

    DoubleVec getCpDerived();
    // Get this control node's reference position in world coordinates.

    DoubleVec getCpOffset();
    // Get this control node's offset vector in world coordinates.

    RWBoolean getBooleanFlag( CntrlNodeFlag flag );
    void setBooleanFlag( CntrlNodeFlag flag, RWBoolean value );

    long getAttribute( CntrlNodeAttr attr );
    void setAttribute( CntrlNodeAttr attr, long value );
};

```

The key members of the class *AdditionCntrlNode* for the addition offset method (derived from the class *BasicControlNode*) are:

```

class AdditionCntrlNode : public BasicControlNode {

public:

    DoubleVec getCpFinal();
    // Calculate a control node's final position via the addition offset method.
};

```

The addition offset method applies vector addition to calculate the final position of the control node rather than the virtual calculation *getCpFinal*.

The key members of the class *TangentCntrlNode* for the tangent plane offset method (derived from the class *BasicControlNode*) are:

```
class TangentCntrlNode : public BasicControlNode {
    private:
        DoubleVec Uij, Vij, Nij;
        // Tangent plane is defined by the two partial derivative: Uij, Vij;
        // Nij is the normal to this tangent plane, i.e. Uij X Vij.

        void setTangentPlane( DoubleVec uij, DoubleVec vij );
        // Set the tangent plane for this offset method.

    public:
        TangentCntrlNode( DoubleVec vector, DoubleVec du, DoubleVec dv );
        TangentCntrlNode( const TangentCntrlNode& );
        // Constructor

        TupleVec getTangentPlane();
        // Get the tangent plane for this offset method.

        DoubleVec getCpFinal();
        // Get a control node's final position in the world coordinate frame.
};
```

This class calculates the position of a CV using a local coordinate frame taken from the parent overlay and an offset interpreted as a position in that local frame.

The key members of the class *FrameOffset* for the skeletal frame offset method (derived from the class *BasicControlNode*) are

```
class FrameOffset : public BasicControlNode {
    private:
        STransformLinHandle *frame;
        // Store the frame.

    public:
        FrameOffset( int dim = 3 );
        FrameOffset( DoubleVec vector );
        // Default constructor containing an identity transformation matrix
        // of the supplied dimension.

        FrameOffset( const DoubleGenMat& basis, DoubleVec vector );
```



```

FrameOffset( const DoubleGenMat& basis, const DoubleVec& origin,
             DoubleVec vector );
FrameOffset( const FrameOffset& v );
FrameOffset( STransformLinHandle *x, DoubleVec vector );
FrameOffset( const STransformLinHandle& x, DoubleVec vector );
// Constructor

void setBasis( const DoubleGenMat& newBasis );
// Use this new matrix as the basis matrix for this frame.
// Here the basis matrix is for a linear transformation, not a spline basis.

void setOrigin (const DoubleVec& newOrigin);
// Use this new DoubleVec as the origin for this frame.

DoubleGenMat getBasis (void);

DoubleVec    getOrigin(void);
// Return the basis vectors and the origin.

inline STransformLinHandle* getFrame();
// Get the frame for this control node.

DoubleVec getCpFinal();
// Apply the linear transformation to basic control node and
// get the control node's final position.
};

```

This class provides the coordinate frame and the method for getting the control node final position in a global coordinate frame.

The key members of the class *GeneralOffset* for the user-defined dynamic offset method (also derived from the class *BasicControlNode*) are:

```

class GenericOffset : public BasicControlNode {

private:

    FuncFromFunction *function;

public:

    GenericOffset( FuncFromFunction *f, int dimension=3 );
    GenericOffset( FuncFromFunction *f, DoubleVec vector );

    DoubleVec getCpFinal();
    // Apply the generic function to the basic control node and
    // get the control node's final position.
};

```

The class *FuncFromFunction* is a user-defined function based on a *C++* function.

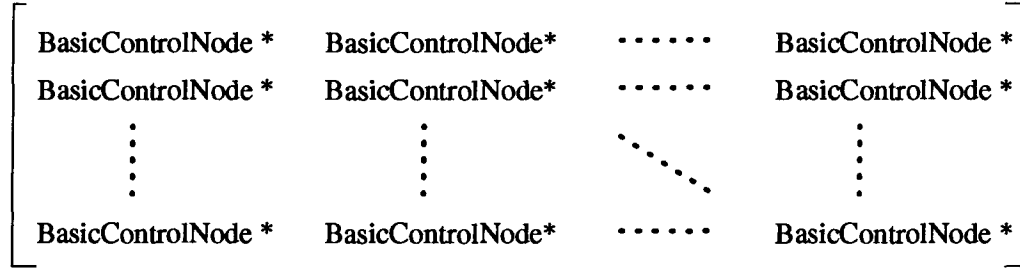


Figure 3.23: The Control Node Matrix Structure

The class *CntrlNodeMatrix* represents the control node mesh of a B-spline surface. In the class *CntrlNodeMatrix*, each element is a control node pointer (Figure 3.23). This class is used for space- and time-efficiency, especially when updating the offset vector or reference position for a control node. All the references to this control node object share the same data, thus it is not necessary to update the different control node objects representing the same control node. Operations on a control node include obtaining its *CpDerived* reference vector ($\vec{R}_{i,j}^{[k]}$), *CpOffset* offset vector ($\vec{O}_{i,j}^{[k]}$), *CpFinal* coordinates ($\vec{V}_{i,j}^{[k]}$), and its attributes (such as moveable, visible and colour).

CntrlNodeMatrix also provides basic operations on the matrix, such as:

- how to set one element in the control node matrix,
- how to update its offset vector in an element,
- how to get its subMatrix,
- how to insert/delete the new rows or columns in a control node matrix,
- how to get an element, row number, and column number.

The above are primitive classes which are used in the hierarchical surface design. The next chapter will examine the space- and time-efficiency for basic operations provided by each of those classes.

3.4 Design Review

In the above, two kinds of quadtree data structures have been proposed as representations for hierarchical B-splines surfaces. Both of these maintain the nature of the surface hierarchy. It is easy to understand this design and implement hierarchical B-splines surfaces via such quadtree data structures. The *C++* classes for hierarchical splines in our design have the following characteristics:

- multiple surface support
- better offset structure
- rational/non-rational uniform refinement (midpoint subdivision)
- compatible with the spline formulation for curves, surfaces and volumes

This system can be used to create a new hierarchical surface given the size, degree, and CV mesh. That is, it supports arbitrary order B-splines, and has the ability to increase/decrease the number of rows/columns in the root-level surface.

Our design and implementation is still incomplete at present. For example, only midpoint subdivision is used. Whenever multi-level editing³ is executed, all affected control nodes in finer overlays are updated immediately. From the time-efficiency point of view, we could buffer edited control nodes at a certain level, then update their offsets at that level and all affected control nodes in finer overlays at the same time when necessary. This would save a lot of time when doing multi-level editing in a hierarchical surface. In the next chapter, this point will be further discussed during the examination of performance.

³When editing takes place at one level overlay of surface definition, any finer overlays resting within the edited area will follow editing changes, which amounts to saying that their control vertices will move in accord with the movement of the edited CV. Also refer back in Section 2.1.

Chapter 4

Storage and Performance Analysis

One of the primary motivations for using a quadtree data structure to represent hierarchical B-spline surfaces is to reduce the storage requirement via the use of aggregation of homogeneous quadtree nodes. This chapter assesses the space- and time-efficiency of this design.

4.1 Storage Benchmarks

This section examines storage costs of two quadtree representations for a hierarchical surface and compares them in the best/worst case, respectively. The “average” storage requirement for representing a hierarchical surface will not be fully explored because of the difficulty of determining what the “average” case is. To evaluate the quadtree representation, the two approaches will be compared with each other, and with the representation used in the prototype hierarchical surface editor. Bi-cubic B-spline surfaces are taken as an example to assess storage costs for these representations.

In the prototype hierarchical surface editor, it is necessary to represent six neighbours of a control node. If we define that each pointer needs $M_{pointer}$ bytes of memory, this costs a fixed $6M_{pointer}$ bytes of memory (6 pointers to north, south, east, west, up and down neighbour, respectively). Suppose that N_{CV} is the number of control nodes in a hierarchical surface, in total, the storage overhead for a hierarchical surface is $6M_{pointer}N_{CV}$ bytes.

4.1.1 Storage Analysis in the Single-overlay Quadtree Representation

The *single-overlay quadtree* approach uses a separate quadtree to represent each level of overlay in the hierarchical surface. The number of nodes required for each quadtree (i.e., for each level of overlay) depends upon the extent that the level has been defined. When a new level is initiated, the quadtree nodes at different levels are generated. As further refinement occurs at this level, every four sibling leaf nodes in the quadtree that contain only non-null references to CV's are coalesced into a parent node that contains a CV matrix containing all of the pointers to CV's in its children. This parent node is converted into a leaf node and its children are deleted. Such processing proceeds until no four homogeneous sibling leaf nodes exist in the quadtree. When this level is fully refined (i.e., all possible CV's are generated), the quadtree evolves to one that has only a single root node containing a CV matrix with all the CV's at that level. Because each node in the quadtree contains five pointers (4 pointers to its children and 1 pointer to its parent), it needs $5M_{pointer}$ bytes of memory in total. Here, an internal node in the quadtree does not contain a CV matrix but simply contains 5 pointers. Each leaf node in the quadtree holds a CV matrix in addition to 5 pointers. The size of this CV matrix is initially 1×1 but as nodes are coalesced (described above), this matrix can grow to a size that encompasses all the CV's at this level. If there are N_{node} quadtree nodes for the entire hierarchical surface, the storage overhead for quadtrees alone is $5M_{pointer}N_{node}$ bytes.

In the *single-overlay quadtree* representation, the storage requirement is minimal when a hierarchical surface is fully refined. In that case, if we define that *level* is the number of overlays in a hierarchical surface, the storage requirement is $N_{CV}M_{pointer} + 5M_{pointer} * level$ bytes because each *single-overlay quadtree* contains only one node. If the hierarchical surface is not fully refined, it is necessary to add some internal nodes in order to reach all

leaf nodes from the root node. Each internal node needs $5M_{pointer}$ bytes of space. This storage cost depends upon refinements of the hierarchical surface. Suppose that N_u is the number of CV's in the u parametric direction at the root level, N_v is the the number of CV's in the v parametric direction at the root level. In the worst case, when every lowest internal node has three leaf nodes, the storage overhead is $5M_{pointer} * \frac{13}{12} 4^{level + \lceil \log \max(N_u, N_v) \rceil}$ bytes.

In the *single-overlay quadtree* representation, if an initial input surface does not contain the $2^n * 2^n$ CV mesh required by a *regular decomposition*, the size of the CV matrix needs to be stored in every node of the quadtree so that during a refinement operation, it is possible to tell when a leaf node becomes a candidate for merging with its sibling nodes. Though such information can be calculated, two extra fields in each node will save a lot of execution time. Suppose that $M_{integer}$ represents the size of an integer, this requires $2M_{integer}N_{node}$ bytes extra space for *single-overlay quadtrees* representing a hierarchical surface if each field needs $M_{integer}$ bytes of memory.

4.1.2 Storage Analysis in the Multiple-overlay Quadtree Representation

In the *multiple-overlay quadtree* approach, a single quadtree stores all the CV's for the entire hierarchical surface. The storage requirement for a *multiple-overlay quadtree* is $5M_{pointer}N_{node}$ bytes if there are N_{node} quadtree nodes in this *multiple-overlay quadtree*. The number of nodes required for such a quadtree depends upon how many patches are in all levels of a hierarchical surface. When a new level initiated, a number¹ of new quadtree nodes are generated. Every refinement at this level generates more nodes. Each leaf node in the quadtree holds one 4×4 CV matrix defining one bi-cubic B-spline patch. Thus, it will require $16M_{pointer} + 5M_{pointer}$ bytes of memory for each leaf node. To speed up processing and evaluation in the "leaf" node, the CV matrix in each internal

¹4, 8, or 16, depending upon where refinement occurs

level	No. of control nodes	No. of overhead pointers in		
		original editor	single quadtree	multiple quadtree
0	16	96	23	21
1	25	150	32	84
2	49	294	56	336
3	121	726	128	1344
4	361	2166	368	5376
5	1225	7350	1232	21504
6	4489	26934	4496	86016
7	17161	102966	17168	344064
8	67801	406806	67808	1376256
9	265225	1591350	265232	5505024
10	1054729	6328374	1054736	22020096

Table 4.2: Storage overheads for a fully refined surface in three kinds of data structures

node is retained (defining a patch at other than the lowest level). A storage overhead of $21M_{pointer}N_{node}$ bytes exists just for the *multiple-overlay quadtree*. In such a data structure, the worst case for storage overhead occurs when a *multiple-overlay quadtree* is full, i.e., each level in the hierarchical surface is fully refined. Each control node requires nearly $21M_{pointer}$ bytes of extra space because each control node affects almost 16 patches of a hierarchical surface and N_{node} almost equals N_{CV} .

In the *multiple-overlay quadtree* representation, if an initial input surface does not contain $2^n * 2^n$ patches required by a *regular decomposition*, it is necessary to add some virtual patches² (implicitly) in the initial input surface to make regular decomposition possible in the patch domain. That is to say, some phantom internal nodes³ are created to reach the root overlay. The memory cost for this is low because it does not add any extra fields into the nodes in the quadtree.

²Virtual patch means it is undefined in the initial surface.

³A phantom internal node is an internal node which does not contain a CV matrix.



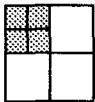
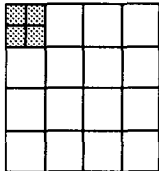
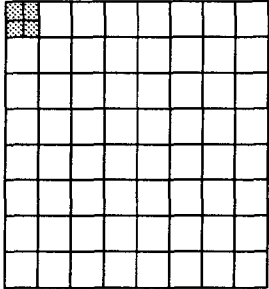
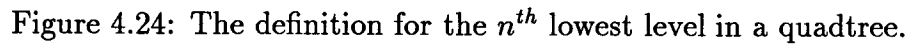
level	No. of CV	representative case	No. of overhead pointers in each level		
			prototype editor	single quadtree	multiple quadtree
0	16		96	23 (1.44 / CV)	21 (1.31 / CV)
1	25		150	32 (1.28 / CV)	84 (3.36 / CV)
2	25		150	133 (5.32 / CV)	84 (3.36 / CV)
3	25		150	140 (5.6 / CV)	84 (3.36 / CV)
4	25		150	147 (5.88 / CV)	84 (3.36 / CV)
5	25	150	154 (6.16 / CV)	84 (3.36 / CV)
6	25	150	161 (6.44 / CV)	84 (3.36 / CV)
7	25	150	168 (6.72 / CV)	84 (3.36 / CV)
8	25	150	175 (7 / CV)	84 (3.36 / CV)
9	25	150	182 (7.28 / CV)	84 (3.36 / CV)
10	25	150	189 (7.56 / CV)	84 (3.36 / CV)

Table 4.3: Storage overheads for a sparsely refined surface in three kinds of structures



Comparing the storage requirements of the two quadtree representations, there is no doubt that a *single-overlay quadtree* surface representation is the best when a hierarchical surface is fully refined or when the patches of a hierarchical surface are clustered. In this case, the *multiple-overlay quadtree* representation requires a lot more storage than the *single-overlay quadtree* representation. On the other hand, the best case for the *multiple-overlay quadtree* representation (when all patches in a hierarchical surface are scattered in different areas) is the worst case for the *single-overlay quadtree* representation, since it will contain a large number of phantom internal nodes. As shown in Table 4.3, when the patches of a hierarchical surface are extremely sparse or scattered, the *single-overlay quadtree* representation has the highest storage overhead.

⁴This is almost the worst case for *single-overlay quadtrees*.

depth of a quadtree (as shown in Figure 4.24) and satisfies the relationship:

$$2^{n-1} * 2^{n-1} - \#N_{CV} < 7 * \#N_{node},^5$$

(i.e., the cost of the quadtree nodes exceeds the number of null entries in all the children) all the subtrees are coalesced into one $2^{n-1} \times 2^{n-1}$ CV matrix. This matrix may contain some null entries corresponding to undefined CV's, but the number of null entries will be less than the total storage required for all of the descendant nodes generated according to homogeneousness. Another way to avoid the worst case would be to utilize both the *single-overlay quadtree* and *multiple-overlay quadtree* representations, i.e., when a hierarchical surface is sparsely refined, adopt the *multiple-overlay quadtree* representation; otherwise, use the *single-overlay quadtree* representation. These methods have not been implemented, and are the subject of future work.

4.2 Performance Benchmarks

This section describes and analyzes the performance of representative operations on a hierarchical surface stored in two kinds of quadtree data structures.

To get an intuitive feeling of their performance, primitive operations in a hierarchical surface were tested to give their execution times when using one of the two different quadtree data structures. At the same time, we analyzed where the execution time was spent in both quadtree representations.

The following primitive operations on a hierarchical surface were examined:

- CV navigation.
 - getting the CV at a given position,
 - getting the north/south/east/west neighbour of a given CV,

⁵Here, $\#N_{CV}$ is the number of generated CV's in this node's CV matrix; $\#N_{node}$ is the number of homogeneous nodes for this node's descendants.

- getting neighbours of a given CV within a certain range,
- getting the CV at the corresponding parametric location on a different level overlay for a given CV.
- Various traversals
 - traverse one level overlay with a given function.
 - traverse an entire hierarchy with a given function.
- Evaluate the surface at a given point
- Refinement
- Multi-level editing

4.2.1 CV Navigation

Getting the control node with given indexes ($level, i, j$) from a hierarchical surface is a basic operation, and its execution time ⁶ in a quadtree data structure is compared with the execution time of retrieving a control vertex with given indexes (i, j) from a *TupleMatrix* (a standard two dimensional array where each element is a vector).

- Execution time for getting a control node from a *TupleMatrix* is 7 microseconds;
- Execution time (in microseconds) for getting a control node from a hierarchical surface in the *multiple-overlay quadtree* representation is shown in Table 4.4.
- Execution time⁷ (in microseconds) for getting a control node from a hierarchical surface in the *single-overlay quadtree* representation is shown in Table 4.5.

⁶The execution time was measured on a Silicon Graphics IRIS 4D crimson workstation.

⁷This result comes from a sparsely refined hierarchical surface (the worst case).

Overlay level	1	2	3	4	5	6	7	8	9	10
Execution time (μsec)	34	39	40	60	72	82	90	101	112	122

Table 4.4: Execution time for getting a CV in the *multiple-overlay quadtree*

Overlay level	1	2	3	4	5	6	7	8	9	10
Execution time (μsec)	43	44	47	70	78	87	101	112	122	131

Table 4.5: Execution time for getting a CV in *single-overlay quadtrees*

It will take more time to get one control node from a hierarchical surface than from a *TupleMatrix* even when the control node lies in the root level overlay, because there are some extra function calls and pointer operations (e.g., getting the quadtree from the hierarchy, getting the root node from the quadtree, getting the control node matrix from the root node, and getting the control node from the matrix).

Table 4.4 and Table 4.5 show that when going to a finer level to search for the control node, an extra 10 microseconds might be required. Most of the extra time is used by bit-shift operations to get the proper node in a quadtree for the given indexes $(level, i, j)$. In a *multiple-overlay quadtree*, execution time has a linear relationship with *level* of the control node. The upper bound on execution time for getting a control node is $O(level)$ in the *multiple-overlay quadtree* representation.

If we define that T_{level} is execution time for getting the proper quadtree representing a certain *level* overlay; *depth* is the level of the quadtree that the node, where the CV at $(level, i, j)$ is contained, lies in, then in the *single-overlay quadtree* representation, the time required to get the control node $(level, i, j)$ is bounded by $O(depth) + T_{level}$. It is necessary to first go to the quadtree representing the *level* (which costs T_{level} execution time); and then in the corresponding quadtree, it is necessary to go to the node that

Level	CV	North CV	South CV	West CV	East CV
1st	34	40	27	36	33
2nd	39	40	30	35	37
3rd	40	35	29	35	31
4th	60	64	28	43	51

Table 4.6: Execution time (in μsec) for getting a CV and its neighbour in the *multiple-overlay quadtree* representation

contains the required control node. This takes $O(depth)$ execution time. In total, its execution time is bounded by $O(depth)^8$.

For comparison, if a single *TupleMatrix* is used to represent each level of a hierarchical surface, approximately $3 \mu sec$ is required to obtain a proper *TupleMatrix* (i.e., a proper level) and $7 \mu sec$ is required to retrieve a CV from the *TupleMatrix*. In total, it takes about $10 \mu sec$.

Getting the north/south/west/east neighbour of a control node with given indexes $(level, i, j)$ is another primitive operation in a hierarchical surface. This takes almost the same execution time as getting the control node with given indexes $(level, i, j)$ except that it has one extra “add” operation ($i + 1$ or $i - 1$ or $j + 1$ or $j - 1$). For both quadtree representations, this point is the same. Table 4.6 shows the execution time for getting a control node and its north/south/west/east neighbour in the *multiple-overlay quadtree* representation of a hierarchical surface. Table 4.7 shows the execution time for getting a control node and its north/south/west/east neighbour in the *single-overlay quadtree* representation of a hierarchical surface.

Getting the control node(s) at the corresponding parametric location of a different level overlay from a given CV $(level, i, j)$ is an elementary operation. No matter what kind of quadtree representation is adopted, this has the same upper-bound execution time as

⁸ T_{level} is far less than $O(depth)$ and ignored.

Level	CV	North CV	South CV	West CV	East CV
1st	43	45	47	47	48
2nd	44	49	48	46	50
3rd	47	51	52	51	53
4th	70	78	74	76	75

Table 4.7: Execution time (in μsec) for getting a CV and its neighbour in the *single-overlay quadtree* representation

Level	position	CV	CV in the <i>parent</i> patch	CV in the <i>child</i> patch
1st	(2,1)	34	-	60
2nd	(1,1)	39	70	65
3rd	(2,2)	40	364	60
4th	(2,2)	60	366	-

Table 4.8: Execution time (in μsec) for getting a CV and the CV's in its *parent/child* patch in the *multiple-overlay quadtree* representation

getting the control node with given indexes ($level, i, j$) except that some extra calculation is required to get the corresponding control node index(es) ($level_n, i_n, j_n$) in the *parent* or *child* patch of a given CV. Table 4.8 shows the execution time for getting a control node and the corresponding CV(s) in the *parent* and *child* patches of a hierarchical surface using the *multiple-overlay quadtree* representation.⁹

Another often-used operation is to get neighbours of a given control node ($level, i, j$) within the index range $[(i_{min}, j_{min}), (i_{max}, j_{max})]$. Suppose the execution time for getting one control node with given indexes ($level, i, j$) is T_{CV} . Then, its upper bound execution time will not be more than $O((i_{max} - i_{min})(j_{max} - j_{min})T_{CV})$ using either of the two quadtree data structures.

⁹Table 4.8 has two testing cases which take 364 and 366 μsec execution time, because the control node lists are obtained in these two testing cases instead of one control node. The control nodes in the control node list form a convex hull containing the given CV.

All of the above operations involve a CV navigation.

4.2.2 Traversals in a Hierarchical Surface

Next, we describe traversal, another kind of elementary operation in a hierarchical surface. In our design, we can traverse an entire hierarchy or one level overlay of a hierarchy with a given function.

In the *multiple-overlay quadtree* representation, execution time for traversing an entire hierarchy has a linear relationship with the number of nodes in the quadtree because every node needs to be visited and has the given function applied to it. Execution time for traversing a certain level of a hierarchy represented by a *multiple-overlay quadtree* includes visiting all the nodes above this level (since it is necessary to first visit nodes above the level in order to reach nodes in the desired level) and applying the given function to all the nodes at this level. Assume that N_{node} is the number of nodes in a *multiple-overlay quadtree*, $N_{nodeAboveLevel}$ is the number of nodes above the specific level of a hierarchy in a *multiple-overlay quadtree*, $N_{nodeInLevel}$ is the number of nodes at the specific level of a hierarchy in a *multiple-overlay quadtree*, and T_{func} is execution time for applying the given function to each node. Then execution times for traversing an entire hierarchy or one level of a hierarchy are bounded by $O(N_{node}T_{func})$ and $O(N_{nodeAboveLevel} + N_{nodeInLevel}T_{func})$, respectively.

In the *single-overlay quadtree* representation, the execution time for traversing an entire hierarchy is related to the number of nodes and the number of leaf nodes in the quadtrees because it is necessary to visit every node and apply the given function to every leaf node. Assume that N_{node} is the number of nodes in all *single-overlay quadtrees*, N_{leaves} is the number of leaf nodes in all *single-overlay quadtrees*, $N_{leavesInOneTree}$ and $N_{nodeInOneTree}$ is the number of leaf nodes and the number of all nodes in one single quadtree representing a certain level overlay of a hierarchical surface in the *single-overlay*

quadtree representation. Then execution time for traversal in a specific level overlay of a hierarchy represented by a *single-overlay quadtree* data structure includes the execution time to reach the proper *single-overlay quadtree* describing that *level* overlay and to traverse this *single-overlay quadtree*. Execution times for traversal in an entire hierarchy or one level overlay of a hierarchy are bounded by $O(N_{node} + N_{leaves}T_{func})$ and $O(level + N_{leavesInOneTree}T_{func} + N_{nodeInOneTree})$, respectively.

4.2.3 Evaluation

Evaluating a point in a hierarchical surface is also a primitive operation (including normal vector and curvature evaluation). Because extra storage has been used to retain the $\vec{R}^{[k]}$ at each level, in the *multiple-overlay quadtree* representation, execution time for evaluating a point is equal to the execution time required to reach the leaf node that contains this point plus the execution time required to evaluate a point in a patch (which is defined by the leaf node).

In the *single-overlay quadtree* representation, it is necessary to first get a CV matrix which defines a patch containing the point. Such an operation may involve several nodes of a quadtree, thus having a longer execution time than when using the *multiple-overlay quadtree* representation. In total, when using the *single-overlay quadtree* representation, the execution time for evaluating a point is equal to the execution time for getting the required CV matrix plus the execution time for evaluating the point in the patch defined by the CV matrix.

For this kind of evaluation operation, the *multiple-overlay quadtree* representation can be better than the *single-overlay quadtree* representation.

4.2.4 Refinement

Refinement is a crucial operation to hierarchical surfaces. The execution time for refinement of a hierarchical surface in two different quadtree data structures will be analyzed.

First, it takes some time to judge whether a given CV $(level, i, j)$ can be refined in the current *level* overlay. If not, it is necessary to go to the parent level to get the corresponding point, repeating this procedure until a proper refinable point is found. Such judgement involves deciding whether at most

$$(2\lceil[u_order/2]/2\rceil + u_order \bmod 2)(2\lceil[v_order/2]/2\rceil + v_order \bmod 2)$$

patches exist at the current level. Here, *u_order* and *v_order* are the B-spline orders in the *u* and *v* direction of parametric space, respectively.

After the proper refinable point in a hierarchical surface is obtained, we need to refine a surface that contains at most

$$(2\lceil[u_order/2]/2\rceil + u_order \bmod 2)(2\lceil[v_order/1]/2\rceil + v_order \bmod 2)$$

patches. During this refinement, space has to be allocated for every recently created control node in the finer level if it does not already exist in the finer level overlay. In the worst case, it is necessary to allocate space for $(2(2\lceil[u_order/2]/2\rceil + u_order \bmod 2) + u_order - 1)(2(2\lceil[v_order/2]/2\rceil + v_order \bmod 2) + v_order - 1)$ control nodes, and set pointers to those objects appropriately.

All of the above steps in both quadtree surface representations have to be executed, and thus both representations have almost the same execution time for these operations.

To decide whether to-be-created control nodes have existed in the lower level overlay takes some time in the *multiple-overlay quadtree* representation because each control node might lie in $u_order * v_order$ patches, i.e., $u_order * v_order$ nodes of a *multiple-overlay quadtree*. Average judgement involves $(u_order * v_order)/4$ nodes of a *multiple-overlay quadtree*. However, this execution time still has a constant upper bound $O(1)$,

although this constant is a little larger. In the *multiple-overlay quadtree* representation, one refinement operation takes between 0.5 and 1 seconds¹⁰.

In the *single-overlay quadtree* representation, one additional step for refinement is still required, i.e., merging every cluster of four nodes into one larger node in the quadtree. In this step, each newly-created node has to check whether it can be merged with three neighbours. If it can be merged, it is necessary to change its phantom parent node to a leaf node, set the corresponding control node pointer matrix properly, and delete the original four leaf nodes in the quadtree. This procedure is repeated for the newly-created leaf node until it can not be merged. This recursive step will take considerable execution time.

For a refinement operation, execution time in the *multiple-overlay quadtree* representation is less than in the *single-overlay quadtree* representation.

4.2.5 Multi-level Editing

Editing a hierarchical surface interactively is an important task in our design. Hierarchical B-spline refinement and offset-referencing provide a mechanism that allows manipulation of the surface *regardless of any previous refinement of the surface*. A change to the surface at any level of refinement closer to the root (level 0 through level $k - 1$) changes $\vec{R}^{[k]}$ altering the shape of the surface. This, in turn, changes the reference information for any recursively defined overlay $S^{[k+1]}$, percolating the effect down through the hierarchy. Similarly, changes to $S^{[k]}$, effected through the offsets at that level, influence the shape of the surface $S^{[k]}$ and all finer levels of overlay within the affected region. In our implementation, this is a non-trivial operation because each affected control node (for which $\vec{O}^{[i]}$ or $\vec{R}^{[i]}$ has been changed) lying in a non-leaf node of the quadtree will affect $(u_order + 1) * (v_order + 1)$ control nodes in the next finer level. If a control node

¹⁰Run on a Silicon Graphics IRIS 4D crimson workstation.

close to the root level is edited and a hierarchy contains a lot of overlays which might be affected, it will take considerable time to update an entire hierarchical surface. This operation in the two kinds of quadtree surface representations has the same execution time bound because each edited CV will affect the same number of CV's in the finer level overlays.

It is better to adopt the “lazy evaluation” method to deal with this situation in order to speed up execution. That is to say, some flag could be set to indicate which control nodes need updating in the future, but those control nodes are not actually updated until they are needed.

Up until now, we have roughly examined the space/performance efficiency in our object-oriented design for hierarchical B-spline surfaces in two different quadtree data structures. The next chapter will look at what can be improved in the future.

Chapter 5

Conclusion and Future Work

This chapter briefly summarizes the design for hierarchical surface representations and discusses what can be further studied in the future.

5.1 Conclusion

Surface representations are important in the domains of computer graphics, geometric modelling, image processing, geographic information systems and robotics. The hierarchical overlay surface is proposed as a general, flexible, space-efficient surface representation. It is coupled with a hierarchical data structure, the quadtree, in order to further reduce memory requirements and to keep its time-efficiency via the nature of homogeneous hierarchies. In this thesis, two object-oriented schemes based on quadtree data structures for the representation of hierarchical B-spline surfaces have been presented. The schemes have the following advantages:

- They keep all the characteristics of hierarchical surfaces, such as the reference-plus-offset feature. This represents regions as a series of overlays with different knot spacing allowing the shape of a hierarchical surface to be modified at multiple levels of overlay. It circumvents the effect of knot insertion for the traditional tensor-product surface upon shape manipulation.
- They support multiple surface representations.

- They provide different offset methods.
- They have an ability to focus on the interesting subsets of the data, which results in an efficient representation and improved execution time.
- It is easy to experiment with new types of hierarchical spline surfaces.

The main algorithms for the hierarchical B-spline surface operations and its corresponding quadtree operations have been described in the previous chapters, and their complexity and memory requirements have also been discussed.

Finally, we took advantage of the object-oriented nature in our implementation because we aimed for flexibility, extensibility, portability and re-use in our design and coding. *C++* exhibits those properties and enables larger programs to be structured in a rational way. Therefore, our modelling tools were written in the *C++* programming language on SGI workstations at the GraFiC/Imager lab.

5.2 Future Work

It is clear that many questions remain. Some of them, by themselves, are topics deserving special attention. Others have known solutions but implementing them can nevertheless be difficult. Thus, it is not surprising that various portions of our modelling system are incomplete. This section briefly describes some of the problems left unsolved and extensions that would be considered useful capabilities within our modelling tools.

One extension is to integrate the non-uniform refinement with the current uniform midpoint subdivision. Thus, the more powerful NURBS could be taken as a mathematical form for representing and designing both standard analytic shapes (conics, quadrics, surfaces of revolution, etc.) and free-form shapes precisely since NURBS are genuine generalizations of non-rational B-spline forms as well as rational and non-rational Bezier

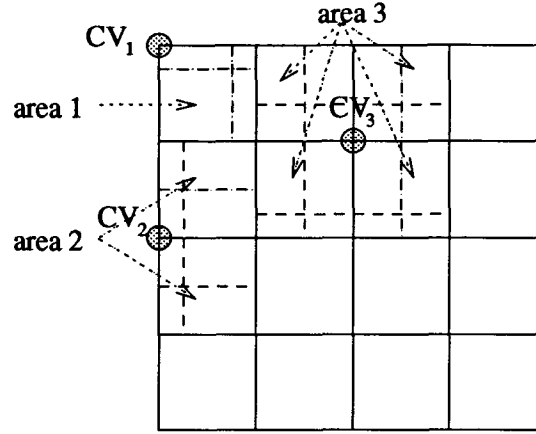


Figure 5.25: One non-uniform refinement case in the *single-overlay quadtree* structure surfaces. In the *multiple-overlay quadtree* representation, this extension is easy to implement to a certain degree since we only need to add two extra fields to each node in the quadtree in order to present non-uniform knots for each patch. However, a lot of difficulty still exists. For example, when there is a *multiple-overlay quadtree* representing a hierarchical surface (Figure 5.25), we cannot further refine this surface at the boundaries between areas 1, 2 and 3. In the *single-overlay quadtree* representation, such extension is more difficult. For example, it is hard to deal with the case shown in Figure 5.25 during the aggregation of homogeneous nodes of a quadtree, even if they lie in the finest level overlay. Suppose we want to do refinements at CV_1 , CV_2 and CV_3 according to the area 1, 2 and 3 patterns, respectively.

This design is constructed to easily extend to hierarchical volumes using octrees. This is an interesting area worth studying further.

Besides these, there are still a number of interesting and attractive avenues for future work. For example, an alternative to the quadtree representation is to use a decomposition method that is not regular (i.e., rectangles of arbitrary size rather than squares). This alternative has the potential of requiring less space via coalescence in a quadtree.

Another interesting topic is the extension of our design to Box-splines, which have

a triangular grid in parametric space. The type of quadtree used often depends on the grid formed by the image sampling process or surface representation. Square quadtrees are appropriate for square grids and triangular quadtrees are appropriate for triangular grids (relating to Box-spline basis functions).

Thus, there is a lot of interesting work which can still be done in surface representation with hierarchical methods.

Appendix A

Mathematical Background on Tensor-product B-spline Surfaces

A *parametric spline* is defined analytically as a set of polynomials over a *knot vector*. A knot vector is a vector of real numbers, called *knots*, in nondecreasing order; i.e.

$\mathbf{u} = [u_0, u_1, \dots, u_q]$ such that $u_{i-1} \leq u_i$, $i = 1, \dots, q$. A spline of *order* k is a C^{k-2} continuous polynomial of degree at most $k - 1$ on each interval $[u_{i-1}, u_i)$.

The i^{th} B-spline basis function of order k (degree $k - 1$) for the knot vector $[u_i, \dots, u_{i+k}]$ is denoted $B_{i,k}(u_i, \dots, u_{i+k}; u)$ and can be expressed as the following recurrence relation:

$$\begin{aligned} B_{i,k}(u_i, \dots, u_{i+k}; u) &= \frac{u - u_i}{u_{i+k-1} - u_i} B_{i,k-1}(u_i, \dots, u_{i+k-1}; u) \\ &\quad + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} B_{i+1,k-1}(u_{i+1}, \dots, u_{i+k}; u) \\ \forall u_i \leq u < u_{i+1} \text{ and } B_{i,1}(u_i, u_{i+1}; u) &= \begin{cases} 1 & u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

The above equation means that the B-spline of order k in the i^{th} span is the weighted average of the B-splines of order $k - 1$ on the i^{th} and $(i + 1)^{st}$ spans, each weight being the ratio of the distance between the parameter and the end knot to the length of the $k - 1$ spans. The computation of $B_{i,k}(u_i, \dots, u_{i+k}; u)$ involves all the knots from u_i to u_{i+k} but no others, since the width of support is k spans.

The restrictions on the specification of a knot vector are that the same value cannot appear more than k times and that the knots must be in nondecreasing order. If the same

knot value u_i occurs t times (i.e., $u_i = u_{i+1} = \dots = u_{i+t-1}$), where $t \leq k$, the continuity at this knot is reduced by $t - 1$.

Moving through the knot vector, each basis function is nonzero over a successive set of $k + 1$ knots. So, $k + m + 1$ knots define $m + 1$ basis functions that correspond to the $m + 1$ control vertices.

A degree (k, l) tensor-product B-spline surface has the form:

$$S(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_{i,k}(u) B_{j,l}(v) V_{i,j}$$

The control vertices $V_{i,j}$ are arranged in a topologically rectangular array called a *control vertex mesh*. $B_{i,k}(u)$ and $B_{j,l}(v)$ are the univariate B-spline basis functions.

The rational B-spline surface representation has one more degree of freedom, *weight*. The degree (k, l) rational B-spline surface (in 3D) is defined as the map of a tensor-product B-spline surface in 4D:

$$\begin{aligned} S(u, v) &= \sum_{i=0}^n \sum_{j=0}^m B_{i,k}(u) B_{j,l}(v) V_{i,j}^w \\ &= \frac{\sum_{i=0}^n \sum_{j=0}^m B_{i,k}(u) B_{j,l}(v) w_{i,j} V_{i,j}}{\sum_{i=0}^n \sum_{j=0}^m B_{i,k}(u) B_{j,l}(v) w_{i,j}} \\ &= \sum_{i=0}^n \sum_{j=0}^m R_{i,k;j,l}(u, v) V_{i,j} \end{aligned}$$

where $V_{i,j}$ are the 3D control points and

$$R_{i,k;j,l}(u, v) = \frac{B_{i,k}(u) B_{j,l}(v) w_{i,j} V_{i,j}}{\sum_{r=0}^n \sum_{s=0}^m B_{r,k}(u) B_{s,l}(v) w_{r,s}}$$

The $R_{i,k;j,l}(u, v)$ functions are the bivariate rational basis functions.

For some applications, subdivision of a B-spline surface is desirable. Subdivision means that a surface constructed

- from one net of control vertices, $\begin{bmatrix} V_{0,0} & \dots & V_{0,n} \\ \vdots & \vdots & \vdots \\ V_{m,0} & \dots & V_{m,n} \end{bmatrix}$

- weighted by two sets of B-splines, $B_{i,k}$ and $B_{j,l}$
- and defined on two sequences of knots, $\{\bar{u}_i\}_0^{m+k}$ and $\{\bar{w}_j\}_0^{n+l}$, respectively

can be represented in terms of

- a large net of control vertices,
$$\begin{bmatrix} W_{0,0} & \dots & W_{0,n+n^1} \\ \vdots & \vdots & \vdots \\ W_{m+m^1,0} & \dots & W_{m+m^1,n+n^1} \end{bmatrix}$$
- weighted by two refined sets of B-splines, $B_i^{[1]}$ and $B_j^{[1]}$
- and defined on two finer sequences of knots, $\{\bar{v}_i\}_0^{m+m^1+k}$ and $\{\bar{w}_j\}_0^{n+n^1+l}$.

After refinement, the surface becomes

$$S(u^{[1]}, v^{[1]}) = \sum_{i=0}^{m+m^1} \sum_{j=0}^{n+n^1} B_{i,k}^{[1]}(u^{[1]}) B_{j,l}^{[1]}(v^{[1]}) W_{i,j}$$

which is defined by a finer control vertex mesh $W_{i,j}$.

In our implementation, we use the Olso algorithm ([Cohen80, Riesenfeld81, Prautzsch84, Prautzsch85, and Lee85]) for our midpoint subdivision. The re-representation of the surface defined by $(m+1) \times (n+1)$ control vertices $[V]$ as a surface of $(m+m^1+1) \times (n+n^1+1)$ new control vertices $[W]$ is accomplished by two matrices composed of the α coefficients, $[\alpha_{left}]$ and $[\alpha_{right}]$:

$$[W] = [\alpha_{left}][V][\alpha_{right}]^T$$

where

$$[V] = \begin{bmatrix} V_{\delta-k+1, \nu-l+1} & \dots & V_{\delta-k+1, \nu} \\ \vdots & \vdots & \vdots \\ V_{\delta, \nu-l+1} & \dots & V_{\delta, \nu} \end{bmatrix},$$

$$[W] = \begin{bmatrix} W_{\mu-k+1, \lambda-l+1} & \dots & W_{\mu-k+1, \lambda} \\ \vdots & \vdots & \vdots \\ W_{\mu, \lambda-l+1} & \dots & W_{\mu, \lambda} \end{bmatrix},$$

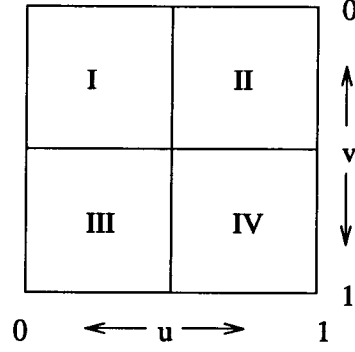


Figure A.26: Regions for four equal patches

$$[\alpha_{left}] = \begin{bmatrix} \alpha_{\delta-k+1,k}(\mu - k + 1) & \dots & \alpha_{\delta,k}(\mu - k + 1) \\ \vdots & & \vdots \\ \alpha_{\delta-k+1,k}(\mu) & \dots & \alpha_{\delta,k}(\mu) \end{bmatrix},$$

$$[\alpha_{right}] = \begin{bmatrix} \alpha_{\gamma-l+1,l}(\lambda - l + 1) & \dots & \alpha_{\gamma,l}(\lambda - l + 1) \\ \vdots & & \vdots \\ \alpha_{\gamma-l+1,l}(\lambda) & \dots & \alpha_{\gamma,l}(\lambda) \end{bmatrix}.$$

The simplest example is derived from the uniform cubic case where each parametric range in u and v is broken at its midpoint. This converts each patch determined by the control vertices $[V]$ into four equal patches according to the diagram (Figure A.26)

$$[\alpha_{left}] = \begin{cases} [A_1] & \text{for regions I and III} \\ [A_2] & \text{for regions II and IV} \end{cases}$$

$$[\alpha_{right}] = \begin{cases} [A_1] & \text{for regions I and II} \\ [A_2] & \text{for regions III and IV} \end{cases}$$

The matrices $[A_1]$ and $[A_2]$ are given by

$$[A_1] = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \end{bmatrix}$$

and

$$[A_2] = \begin{bmatrix} \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{bmatrix}$$

A more detailed treatment of this material, and computational algorithms for B-spline, Bezier, and NURB surfaces can be found in [Bartels87].

Appendix B

The Definition of Quadtrees and Their Characteristics

A quadtree is a data structure, originally used for image representation, that is based on the successive subdivision of the image into rectangular quadrants. It is represented by a tree of outdegree 4¹, where the root corresponds to the image itself and its four children correspond to the northwest, northeast, southwest and southeast quadrants respectively. The root node has no parent and leaf nodes have no children. Each node in the quadtree contains six pieces of information. The first five items are pointers to the node's parent and to its four children. The sixth piece of information describes the node itself (such as colour, etc). Figure 3b is a block decomposition of the region in Figure 3a. Figure 3c is the corresponding quadtree, which encoded image array (Figure 3b). It exploits two-dimensional coherence by recursively decomposing such an image into square areas of identical colour. This decomposition begins with a tree structure consisting of a single root node corresponding to the whole image. Unless the image is homogeneous (i.e., all the same colour), it is subdivided into four quadrants. This process is repeated for any non-homogeneous sub-region. Each of the leaf nodes in the resulting quadtree represent a region having the same colour.

In our design, if the quadtree decomposition is governed by control vertices, each $m_i \times n_i$ CV matrix for one level i when fully-refined in a hierarchical surface is taken as

¹A tree of outdegree n means that each node in the tree has at most n children.

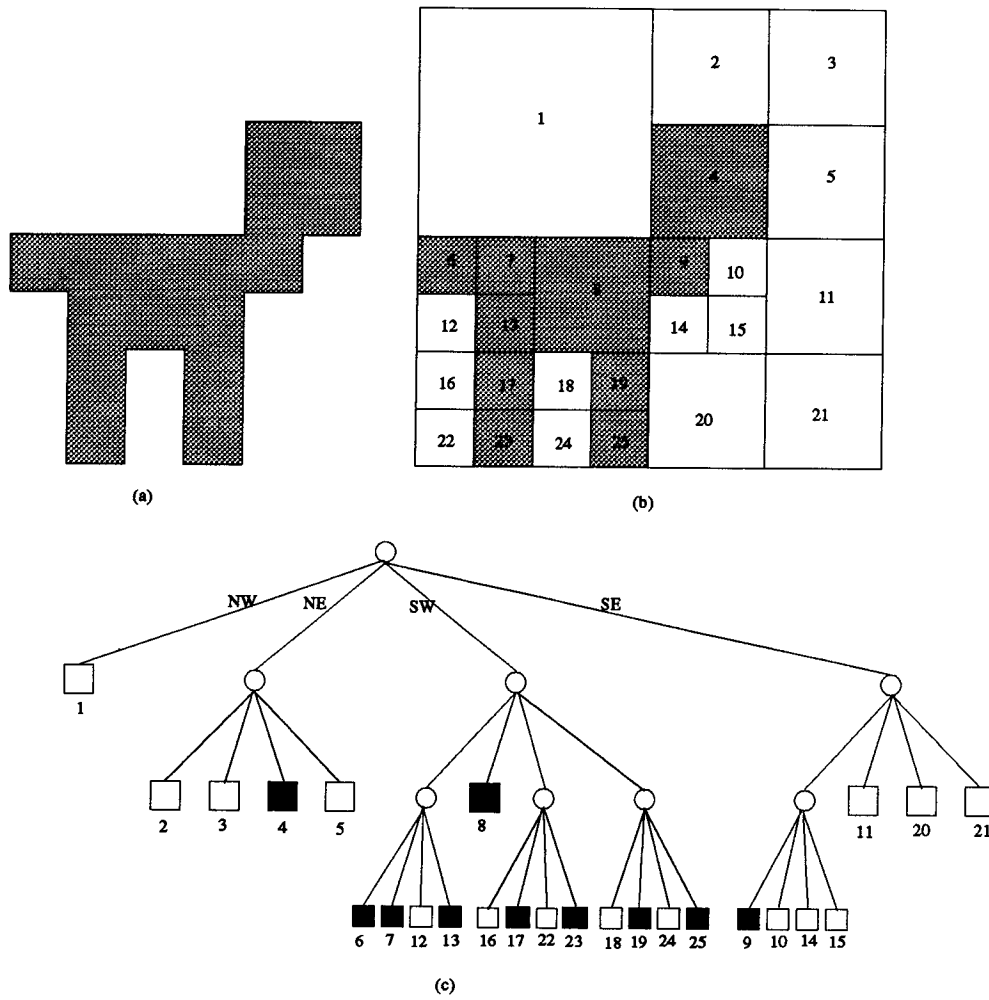


Figure B.27: A region, its maximal blocks, and the corresponding quadtree. (a) Region. (b) Block decomposition of the region in (a). (c) Quadtree representation of the blocks in (b).

an image array; each CV is an element in such an image array. In a hierarchical surface, each overlay may be only a subset of the full $m_i \times n_i$ array of CV's. A homogeneous region in the array is one where either none or all of the CV's are defined. That is to say, the CV matrix is recursively decomposed.

If the quadtree decomposition is governed by patches, each $s_i \times t_i$ patch matrix for one level i when fully-refined in a hierarchical surface is taken as an image array; each patch is an element in such an image array and defined by one CV matrix. In a hierarchical surface, each overlay may be only a subset of the full $s_i \times t_i$ array of patches. A homogeneous region in the array is one where either none or all of the patches are defined. That is to say, patches are recursively decomposed.

There are numerous ways to represent quadrees. These range from fully pointered quadrees in which each pointer from parent to child is stored explicitly, to pointerless quadrees in which no pointers are stored. Mark and Lauzon [Mark85] compare the merits and space-efficiency of several quadtree data structures. Fully pointered quadrees offer maximum flexibility because they can be traversed in any order, but more storage space is required for the pointers. Pointerless quadrees are the most compact in terms of storage requirements, but they have to be traversed in the order of their creation, which reduces the speed of some algorithms. These represent the two extremes for time- and space-efficiency.

In the literature, three types of structures for quadrees are reported. Klinger and Rhodes ([Klinger79]) index nodes using a form of key derived from the ordered list of ancestors of the node, and use size and coordinate information for operations. Hunter ([Hunter79]) applies a "roped net", a pointer-based structure where each node is linked to its neighbours. A simpler hierarchical structure was adopted by Samet et al ([Samet80]). with a node linked only to its parent and children.

Bibliography

- [1] S. K. Abdali and D. S. Wise, "Experiments with quadtree representation of matrices," Proceedings of the International Symposium on Symbolic and Algebraic Computation, July 1988.
- [2] D. Ayala, P. Brunet, R. Juan, and I. Navazo, "Object representation by means of non-minimal division quadtrees and octrees," ACM Transactions on Graphics 4, January 1985, 41-59.
- [3] R. H. Bartels, and John C. Beatty, "An introduction to Splines for use in Computer Graphics and Geometric Modeling," Morgan Kaufmann Publishers, Inc., 1987.
- [4] M. A. Bauer, "Set operations in linear quadtrees," Computer Vision, Graphics and Image Processing 29, February 1985, 248-258.
- [5] D. A. Beckley, M. W. Evens, and V. K. Raman, "Multikey retrieval from k-d trees and quad-trees," Proceedings of the SIGMOD conference, Austin, TX, May 1985, 291-301.
- [6] P. W. Besslich, "Quadtree construction of binary images by dyadic array transformations," Proceedings of the IEEE Conference on Pattern Recognition and Image Processing, Las Vegas, June 1982, 550-554.
- [7] W. Boehm, "Rational geometric splines," Computer Aided Geometric Design 4, 1987, 67-77.
- [8] F. W. Burton, and J. G. Kollias, "Comment on the explicit quadtree as a structure for computer graphics," Computer Journal 26, 2, May 1983, 188.
- [9] F. W. Burton, V. J. Kollias and J. G. Kollias, "Expected and worst-case storage requirements for quadtrees," Pattern Recognition Letters 3, 2, March 1985, 131-135.
- [10] I. Carlbom, I. Chakravarty, and D. Vanderschel, "A hierarchical data structure for representing the spatial decomposition of 3-D objects," IEEE Computer Graphics and Applications 5, 4, April 1985, 24-31.
- [11] C. H. Chien and J. K. Aggarwal, "A normalized quadtree representation," Computer Vision, Graphics and Image Processing 26, 3, June 1984, 331-346.
- [12] C. H. Chien and J. K. Aggarwal, "Reconstruction and matching of 3-D objects using quadtrees/octrees," Proceedings of the Third Workshop on Computer Vision: Representation and Control, Bellaire, MI, October 1985, 49-54.
- [13] Hiroaki Chiyokura & Fumihiko Kimura, "Design of Solids with Free-Form Surfaces," Computer Graphics, Vol. 17, No. 3, July 1983.

- [14] J. H. Chu, "Notes on expected numbers of nodes in a quadtree," Computer Science Department, University of Maryland, College Park, MD, January 1988.
- [15] J. H. Clark, "Hierarchical geometric models for visible surface algorithms," Communications of the ACM 19, 10, October 1976, 547-554.
- [16] E. Cohen, T. Lyche, and R. F. Riesenfeld, "Discrete B-splines and subdivision techniques in computer-aided geometric design and computer graphics," Computer Graphics and Image Processing, 14, 2, October 1980, 87-111.
- [17] Sabine Coquillart, "Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling," Computer Graphics, Vol. 24, No. 4, August 1990.
- [18] M. S. Cottingham, "A compressed data structure for surface representation," Computer Graphics Forum 4, 3, September 1985, 217-228.
- [19] W. A. Davis and X. Wang, "A new approach to linear quadtrees," Proceedings of Graphics Interface 85, Montreal, May 1985, 195-202.
- [20] C. R. Dyer, A. Rosenfeld, and H. Samet, "Region representation: boundary codes from quadtrees," Communications of the ACM 23, 3, March 1980, 171-179.
- [21] C. Faloutsos, T. Sellis, and N. Roussopoulos, "Analysis of object oriented spatial access methods," Proceedings of the SIGMOD Conference, San Francisco, May 1987, 426-439.
- [22] Gerald Fairn, "Curves and Surfaces for Computer Aided Geometric Design: A practical guide," Academic Press, Inc., 1988.
- [23] R. A. Finel and J. L. Bentley, "Quad trees: a data structure for retrieval on composite keys," Acta Informatica 4, 1, 1974, 1-9.
- [24] D. R. Forsey, "Part II: Hierarchical Free-Form Surfaces," Ph.D thesis, Univ. of Waterloo, 1990.
- [25] J. H. Friedman, F. Baskett, and L. J. Shustek, "An algorithm for finding nearest neighbors," IEEE Transactions on Computers 24, 10, October 1975, 260-269.
- [26] D. R. Fuhrmann, "Quadtree traversal algorithms for pointer-based and depth-first representations," IEEE transactions on Pattern Analysis and Machine Intelligence 10, 6, November 1988, 955-960.
- [27] Irene Gargantini, "An Effective Way to Represent Quadtrees," Comm. of the ACM, Vol 25, No. 12, Dec. 1982, 905-910.
- [28] N. K. Gautier, S.S. Iyengar, N. B. Lakhani, and M. Manohar, "Space and time efficiency of the forest-of-quadtrees representation," Image and Vision Computing 3, 2, May 1985, 63-70.
- [29] T.N.T. Goodman, "Shape preserving interpolation by parametric rational cubic splines," Proc. Int. Conf. on Numerical Mathematics, Int. Series Num. Math. 86, 1988, Birkhauser, Basel.

- [30] G. M. Hunter, and K. Steiglitz, "Operations on Image Using Quad Trees," IEEE Trans. Pattern Anal. & Mach. Intell., Vol PAMI-1, 1979, 145-153.
- [31] A. Hunter, and P. J. Willis, "Breadth-first quad encoding for networked picture browsing," Comput. & Graphics, Vol 13, No. 4, 1989, 419-432.
- [32] T. J. Ibbs and A. Stevens, "Quadtree storage of vector data," International Journal of Geographical Information Systems 2, 1, January-March 1988, 43-56.
- [33] C. L. Jachins and S. L. Tanimoto, "Quad-trees, oct-trees, and k-trees - a generalized approach to recursive decomposition of Euclidean space," IEEE Transactions on Pattern Analysis and Machine Intelligence, 5, September 1983, 533-539.
- [34] L. Jones and S. S. Iyengar, "Space and time efficient virtual quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence 6, 2, March 1984, 244-247.
- [35] G. Kedem, "The quad-CIF tree: a data structure for hierarchical on-line algorithms," Proceedings of the Nineteenth Design Automata Conference, Las Vegas, June 1982, 352-357.
- [36] M. L. Kersten and P. van Emde Boas, "Local optimizations of quadtrees," Technical Report IR-51, Free University of Amsterdam, Amsterdam, The Netherlands, June 1979.
- [37] A. Klinger, and C. R. Dyer, "Experiments in Picture Representation Using Regular Decomposition," Comput. Graphics and Image Processing, Vol. 5, 1976, 68-105.
- [38] A. Klinger, and M. L. Rhodes, "Organization and access of image data by areas," IEEE Trans. Pattern Anal. Mach. Intell., 1, 1979, 50-60.
- [39] Koji Komatsu, "Human skin model capable of natural shape variation," The Visual Computer, 3, 1988, 265-271.
- [40] S. X. Li and M. H. Loew, "The quadcode and its arithmetic," Communications of the ACM 30, 7, July 1987, 621-626.
- [41] S. X. Li and M. H. Loew, "Adjacency detection using quadcodes," Communications of the ACM 30, 7, July 1987, 627-631.
- [42] D. M. Mark and J. P. Lauzon, "The space efficiency of quadtrees: an empirical examination including the effects of 2-dimensional run-encoding," Geo-Processing 2, 1985, 367-383.
- [43] D. C. Mason, and M. J. Callen, "Comparison of two dilation algorithms for linear quadtrees," Image and Vision Computing 6, 3, August 1988, 169-175.
- [44] M. Nahas, H. Huitric and M. Saintourens, "Animation of B-Spline Figure," The Visual Computer, 3(4), 1987.
- [45] R. C. Nelson and H. Samet, "A population analysis of quadtrees with variable node size," Computer Science TR-1740, University of Maryland, College Park, MD, December 1986.

- [46] R. C. Nelson and H. Samet, "A population analysis for hierarchical data structures," Proceedings of the SIGMOD Conference, San Francisco, May 1987, 270-277.
- [47] M. A. Oliver, and N. E. Wiseman, "Operations on quadtree encoded images," The Comp. J. 26(1), 1983, 83-90.
- [48] M. A. Oliver and N. W. Wiseman, "Operations on quadtree leaves and related image areas," Comp. J. 26, 4, November 1983, 375-380.
- [49] M. H. Overmars, "Geometric data structures for computer graphics: an overview," in Theoretical Foundations of Computer Graphics and CAD, Springer-Verlag, Berlin, 1988, 21-49.
- [50] F. Peters, "An algorithm for transformations of pictures represented by quad-trees," Computer Vision, Graphics and Image Processing 32, 3, December 1985, 397-403.
- [51] L. Piegl, and W. Tiller, "Curve and Surface constructions using rational B-splines," Computer Aided Design, Vol. 19, 1987, 485-498.
- [52] S. Ranade, A. Rosenfeld and J. M. S. Prewitt, "Use of quadtrees for image segmentation," Computer Science TR-878, University of Maryland, College Park, MD, February 1980.
- [53] W. C. Rheinboldt and C. K. Mesztenyi, "On a data structure for adaptive finite element mesh refinements," ACM Transactions on Mathematical Software 6, 2, June 1980, 166-187.
- [54] R. F. Riesenfeld et al. "Using the Oslo Algorithm as a Basis for CAD/CAM Geometric Modeling," Proc. NCGA National Conf. NCGA, Fairfax, Va. 1981, 345-356.
- [55] D. F. Rogers, and L. A. Adlum, "Dynamic rational B-spline surfaces," Computer Aided Design, Nov. 1990, 609-616.
- [56] H. Samet, "Region representation: quadtrees from boundary codes," Communications of the ACM 23, 3, March 1980, 163-170.
- [57] H. Samet and A. Rosenfeld, "Quadtree structures for image processing," Proceedings of the Fifth International Conference on Pattern Recognition, Miami Beach, December 1980, 815-880.
- [58] H. Samet, "Connected component labeling using quadtrees," J. of the ACM 28, 3, July 1981, 487-501.
- [59] H. Samet, "Neighbor Finding Techniques for Images Representated by Quadtrees," Comp. Graphics and Image Processing, Vol. 18, 1982, 37-57.
- [60] H. Samet, "The quadtree and related hierarchical data structures," Computer Vision, Graphics, and Image Processing 26, 1, April 1984, 187-260.
- [61] H. Samet and R. E. Webber, "On encoding boundaries with quadtrees," IEEE Transactions on Pattern Analysis and Machine Intelligence 6, 3, May 1984, 365-369.

- [62] H. Samet, "A top-down quadtree traversal algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7, 1, January 1985, 94-98.
- [63] H. Samet and C. A. Shaffer, "A model for the analysis of neighbor finding in pointer-based quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7, 6, November 1985, 712-720.
- [64] H. Samet, and R. E. Webber, "Hierarchical data structures and algorithms for computer graphics, Part I Fundamentals," *IEEE Comp. Graphics and Appl.*, May 1988, 48-68.
- [65] H. Samet, and R. E. Webber, "Hierarchical data structures and algorithms for computer graphics, Part II Applications," *IEEE Comp. Graphics and Appl.*, July 1988, 59-75.
- [66] H. Samet, "Design and Analysis of Spatial Data Structures," Addison-Wesley, Reading, MA, 1990.
- [67] D. S. Scott and S. S. Iyengar, "A new data structure for efficient storing of images," *Pattern Recognition Letter* 3, 3, May 1985, 211-214.
- [68] C. A. Shaffer and H. Samet, "Optimal quadtree construction algorithms," *Computer Vision, Graphics, and Image Processing* 37, 3, March 1987, 402-419.
- [69] M. Shneier, "Note: Calculations of Geometric Properties Using Quadtrees," *Computer Graphics and Image Processing*, Vol 16, 1981, 296-302.
- [70] M. Tamminen, "Comment on quad- and oct-trees," *Communications of the ACM* 27, 3, March 1984, 248-249.
- [71] S. Tanimoto and T. Pavlidis, "A hierarchical data structure for picture processing," *Computer Graphics and Image Processing* 4, 2, June 1975, 104-119.
- [72] Tiller, W., "Rational B-splines for curve and surface representation," *IEEE Comput. Graph. Appl.* 3, 9, September 1983, 61-69.
- [73] A. Unnikrishnan, Y. V. Venkatesh, and P. Shankar, "Connected component labelling using quadtrees - a bottom-up approach," *Computer Journal* 30, 2, April 1987, 176-182.
- [74] J. R. Woodward, "The explicit quad tree as a structure for computer graphics," *Computer Journal* 25, 2, May 1982, 235-238.
- [75] J. R. Woodward, "Compressed quad trees," *Computer Journal* 27, 3, August 1984, 225-229.
- [76] M. A. Yerry and M. S. Shephard, "A modified quadtree approach to finite element mesh generation," *IEEE Computer Graphics and Applications* 3, 1, January/February 1983, 39-46.