WOOD WEATHERING SIMULATION WITH 3 DIMENSIONAL SURFACE CELLULAR AUTOMATA

LATHSAMY CHIDTAVONG

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIRMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

SCHOOL OF GRADUADE STUDIES

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2007

หัวข้อวิทยานิพนธ์	การจำลองการเปลี่ยนสีของไม้ ที่มีผลกระทบจากสภาพดินฟ้าอากาศโดยวิธีการแบ่งพื้นผิวด้วยเซลลูล่าออโตมาตาแบบ 3 มิติ
นักศึกษา	นาง รัศมี จิตวงศ์
รหัสประจำตัว	47068013
ปริญญา	วิทยาศาตรมหาบัณฑิต
สาขาวิชา	วิทยาการคอมพิวเตอร์
พ.ศ.	2550
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ. ดร. กรกช ประชุมรักษ์

บทคัดย่อ

วิทยานิพนธ์นี้ มีวัตถุประสงค์เพื่อนำเสนอวิธีการจำลองการเปลี่ยนสีของวัตถุต่างๆที่ทำด้วยไม้ ซึ่งได้รับผลกระทบจากสภาพดินฟ้าอากาศ เนื่องจากไม้เป็นวัตถุที่สามารถเกิดการเปลี่ยนแปลงของสีได้ง่าย โดยเฉพาะบริเวณพื้นผิวซึ่งได้รับผลกระทบโดยตรงจากสภาพดินฟ้าอากาศที่เปลี่ยนแปลงไป อาทิ ไม้ที่ถูกทิ้งไว้ในที่แจ้ง โดยวิธีการที่ได้นำเสนอในงานวิจัยนี้จะเริ่มต้นจากการสร้างแบบจำลองวัตถุไม้ จากนั้นเมื่อได้แบบจำลองที่ต้องการแล้วจึงทำการแปะแบบจำลองนั้นด้วยพื้นผิวไม้ และทำการใช้วิธีการแบ่งพื้นผิวด้วยเซลลูล่าออโตมาตาแบบ 3 มิติ เพื่อกระจายและแสดงปรากฏการณ์การเปลี่ยนสีบนพื้นผิวสามมิติ ในขั้นตอนนี้สีของไม้จะค่อยๆเปลี่ยนไปตามกฎของเซลลูล่าออโตมาตา ซึ่งผลลัพธ์ที่ได้คือ ปรากฏการณ์การเปลี่ยนสีของวัตถุไม้ที่มีความเสมือนจริง

Thesis Title	WOOD WEATHERING SIMULATION WITH 3 DIMENSIONAL SURFACE CELLULAR AUTOMATA
Student	Mrs. Lathsamy Chidtavong
Student ID	47068013
Degree	Master of Science
Program	Computer Science
Year	2007
Thesis Advisor	Asst. Prof. Dr. Korakot Prachumrak

ABSTRACT

This research presents a method to simulate weathering on any wooden objects. Wood is a material that the appearance is changed easily, especially the damage in the surface of the outdoor wood. To produce the realistic simulation in computer graphics, first the wooden model is constructed. After that the wood texture is mapped to the model. Finally the 3D surface cellular automata method (3DSCA) is used to generate and render weathered wood on 3D surface. The cells positions are specified randomly, then they are changed by the specific rules which cause the change in the colors of the wood. At the end, the realistic weathered wood is generated.

ACKNOWLEDGEMENT

I would like to take this occasion to express my genuine gratitude to those who helped, guided or suggest me in this challenging research.

Almost my research successful completeness, because of I had a helpful contribution from many parts. First of all I am very grateful to Asst. Prof. Dr. Korakot Prachumrak, the advisor of my research, for her kindness and helpful advice in choosing methods and techniques, correcting written, also throughout my graduate studies. In addition, I would like to say thanks to Miss. Janejira Chatchawal, the Thai graduate student master degree in computer science in Thai program, for solving problems and helpful advice in programming. Furthermore, I am very thankful to all lecturers for their attendance pass knowledge on me. Their have helped me develop knowledge stronger than before. Also, I would like to say thank to Mr. Pasit Sasikarn and friends, for comment and encouragement all the time. Finally I am very thankful to my organizer and my family for giving me a chance to develop knowledge.

LATHSAMY CHIDTAVONG

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

Page

TABLE OF CONTENTS (Cont)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Cont)

LIST OF FIGURES (Cont)

CHAPTER 1

INTRODUCTION

1.1 Statement and significance of the problems

Computer simulation is one interesting field in computer graphics and is used to mimic several phenomenal in many fields in science, for example in Biology, it is used for modeling the spreading of fungi, in Chemistry, it is used for modeling the configuration of molecules, etc.

The degeneration of wood is an interesting phenomenon and can be simulated with computer graphics. Lately, only a few literary works have been presented on the simulation of weathered wood. Unfortunately, the beautiful patterns of wood were ignored, and only small shapes of wood could be presented. On the other, the techniques used to simulate natural phenomena have been presented, especially the 3 Dimensional Surface Cellular Automata (3DSCA) technique. This technique was used to simulate some natural phenomena that affected on the surface of the objects and gave a realistic result. However, the previous researches used the 3DSCA technique for simulation of the objects that did not have the surface texture, for example: mud, concrete, etc. Thus, this research proposes a method to represent the phenomena of weathering on wooden objects by combination of two techniques. The texture mapping for mapping wood texture on the model and 3D surface cellular automata technique for spreading weathering of wood. The heart of this research is the combination of Perlin noise and cellular automata techniques.

Perlin noise technique is utilised for generating natural texture. This technique has individual functions to generate interesting textures. The several natural patterns of wood are generated by this technique. Cellular automata is a discrete model from the area of theory of computation and mathematics. The simple rules of cellular automata can produce interesting patterns. Also, the change in wood's colors can be altered by cellular automata rules.

1.2 Objectives

The aim of this research is to simulate weathering on wooden objects by combination of the 3DSCA method and texture mapping technique. This research will present the new rules of cellular automata that are applied in the 3DSCA method, and will present the method to generate

wood textures by using Perlin noise technique. These wood textures are applied with texture mapping technique.

1.3 Hypothesis to be tested

The result of this simulation will be the presentation of more realistic weathered wooden objects. Because, the surfaces of the 3D wooden objects are covered by wood textures, and the 3D Surface Cellular Automata (3DSCA) method is used for generating effects on wood, it means the spreading of inflected color is achieved by generation of CA's rules.

1.4 Limitation of the study

This research is in the field of computer graphics that simulates weathering effects of wooden objects. There are many related mathematical functions in the process of reconstruction 3D model, and also in the 3D model there are a lot of cells have to be rendered. As the result, the time consumed for computing is very high. It requires high CPU speed and large memory. Because the computers currently used for this research are not fast enough, this research can not show the large scale objects which consist of a huge collection of triangle faces. According to experiments in this research, the 3D objects that consist of at least 100,000 triangle faces may take a long time to simulate or may hang and can not bring results. The requirements of this research are listed below:

1. 3D model has to be .ase extension file
2. VGA 256 MB or up
3. RAM 1 GB or up
4. CPU 2 GB or up

CHAPTER 2

BACKGROUND

2.1 Literature review

The simulation of natural phenomena using computer graphics has been investigated since long time ago. Many researchers have exposed their researches in different methods. In 1996, an approach for modeling and rendering of one type of weathering metallic patina [7] was represented based on multiple layers approach method. This approach is capable of simulating a variety of metallic patinas. The Kubelka-Munk model is used to model the reflectance and transmission of light through the layered structure. Because of many layers, the time consumed is very high. Weathered stone is a phenomenon simulated in 1999 [8]. In this technique, the flow of water, the transport, and dissolution of minerals within the volume and erosion of the surface are simulated for modeling and rendering of changes in the shape and appearance of stones. Also this method can capture a variety of weathered appearances.

Evaluation of computer simulation continues to be developed. The new form of corrosion based on simple physical characteristics is exhibited [19]. This method uses the random walk technique adapted to a generic model that allows user to predict the evolution of corrosion over time. Another research [5] introduces a novel method to simulate the distortion of an object that is caused by repeated long-term impacts. This works by updating the vertices of an adaptively edited object mesh. Their simulation can apply several impacts in a few seconds. Cracking and Peeling are other interesting revelation, many characteristics of Cracking are demonstrated, such as: cracks and fractures in solid objects [13] and a simulation of paint cracking and peeling [4] Theses approaches provide the tools to control the pattern of cracks where [17, 18] have simulated realistic propagation of various types of peeling and cracking respectively. The simulation used to model fracturing, both in computer graphics and in engineering, is presented by O'Brien and Hodgins [1]. Their fracture model is based on theory of linear elastic fracture mechanics to animating breaking objects. In recent years, a system for modeling lichens, and simulating their propagation and growth in a virtual scene is exhibited by Desbenoit et al [2]. Also, they are able to control the development of lichens. Philippe Even and Stephane Gobron demonstrated many natural phenomena on one building [15]. They used photographs of the building to reconstruct it in the model. After that many kinds of weathering is simulated, such as:

moss, lichen, etc. The last referable research is wood aging. In [21] a visual simulation of the distortion, cracking, and erosion of wood is proposed. Wood in this method is exhibited by a tetrahedral mesh. The surface of the wood is defined by values assigned to the superficial tetrahedral mesh vertices. Changes in the surface are achieved by value changes. This method is suitable for simulating only small objects.

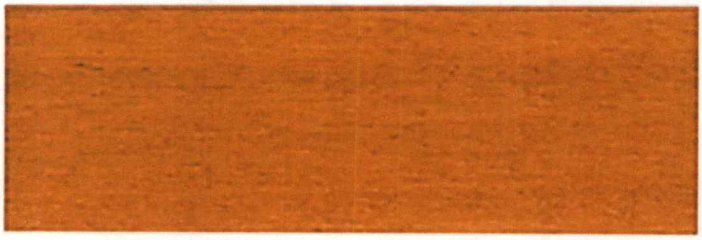
2.2 Background of wood

Wood is a kind of material which has a strong relationship with human beings. Wood is used to produce tools and other objects from small ones to big ones. Tables, chairs, and buildings are some examples. Wood has many distinguished properties such as, endurance, ease of repair, and the beautiful grain patterns and colors. However, each kind of wood has different structure, so the characteristics of colors, weight and texture are all different. Wood composes of two parts namely: Heartwood and Sapwood. Sapwood consists of the outer layers near the cambium that contains both living and dead cells. The structure of the Sapwood is such that it does not have high density and can change easily when affected by weather. The effect slowly spreads from outside to the inside of wood. For the effect, all the wood's characteristic will be changed, especially the colors.

2.2.1 Evolution of weathering

Weathering is the general term used to define the slow degradation of materials exposed to the weather. The degradation mechanism depends on the type of material, but the cause is a combination of factors found in nature: moisture, sunlight, heat/cold, chemicals, abrasion by windblown materials, and biological agents.

Leaving the wooden objects in exposed places for a long time will affect their properties. The location where the wood is placed, is an important element for that effect. Firstly the surface structure will be decomposing. The wood will be slightly discolored, usually first darkening or yellowing, then weathering to a silvery grey if left outdoors long enough. This process means that the soft part is damaged, and after that the hard parts of wood exhibit.



Oiled Kapur Wood (new)

Weathered Kapur Wood

a.

Before

After

b.

Figure 2. 1: The different surfaces of wood before and after weathering

2.3 The 3D standard models

The 3D standard models are the computer graphics test models. They can be used to test various graphics algorithms, including polygonal simplification, compression, and surface smoothing. Usually, the 3D standard models are stored in .ply files. This format was developed at Stanford University, and represented the file in ASCII formats. Choosing ASCII makes it possible for someone unfamiliar with it to get a feel for the file format, and it avoids the problem of using the correct big-endian vs. little-endian byte orders. The 3D standard models can be downloaded from many sources and have many files formats [34-37]. For converting .ply files to other

formats, some converters are illustrated in [34]. The one converter that we will become familiar with, is PLY-to-Maya plugin. It supports importing from PLY to Maya, and exporting from Maya to PLY, for versions 6.0 and 7.0 of Maya.

One of the famous 3D standard models is the Stanford Bunny model. It is most commonly used to test models in computer graphics. It is a collection of 35,947 vertices or 69,451 triangles. The Bunny was created by using a technique that was developed by Greg Turk and Marc Levoy. The bunny is considered to be too good as a test model. It is smooth, has a lot of connectivity, and it isn't too complex. The original photograph of the Stanford Bunny and the two examples of how the Stanford Bunny model is applied to test two theories are shown below:

Figure 2. 2: The color image of the original Bunny model took on April 1, 2003 [38]

Figure 2. 3: Bunny image from paper named "Real-Time Fur over Arbitrary Surfaces" [10]

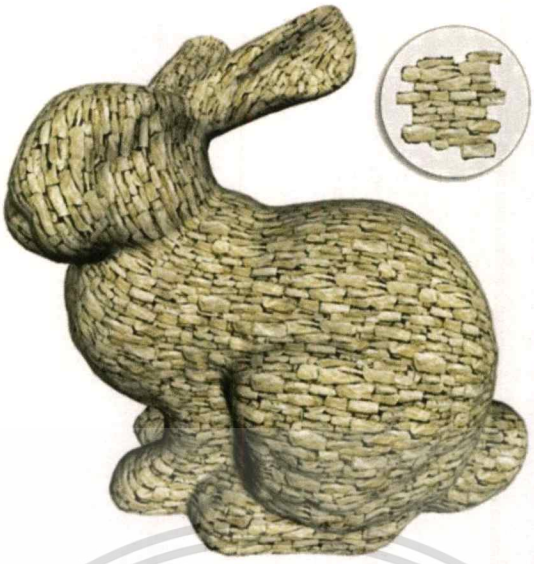


Figure 2. 4: Bunny image from paper named Lapped Textures [6]

2.4 Cellular Automata

2.4.1 Introduction

Cellular Automata (CA) was first introduced by John von Neumann and Ulam in the 1940s who called it Cellular Spaces. After that Wolfram has continuously studied it, as illustrated in a book titled `A New Kind of Science'[12]. The study of cellular automata is mostly on one-dimensional and two-dimensional infinite grids, though higher dimensions are also considered, and has attracted researchers from various disciplines. Cellular automata is applied in different branches of science, Biology, Chemistry, Physics, etc. The unique characteristic of CA is its simple structure. When iterated several times it can produce complex patterns. It displays the potential to simulate different sophisticated natural phenomena. There are many papers [22-24, 26] on the definition of CA in each related areas. However, the definition of CA in [25] as "is a collection of "colored" cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells." is the most suitable for this research. In short, cellular automata consist of a lattice of cells, each of which is in one of a finite number of states at any given time, a neighborhood of nearby cells defined of each cell, and transition rules that describe how a given cell change its state based upon its current state and those of its neighboring cells. Their relationship can be expressed [11] as:

$$S_i(t+1) = f(S_i(t) + S_{neigh(i)}(t)) \qquad \text{(eq 2.1)}$$

Where: S_i denotes the state of the i^{th} cell

t denotes the number of generations that have evolved

$S_{neigh(i)}$ denotes the set of neighbors (according to the chosen neighborhood) of the i^{th} cell

One of the most fundamental properties of cellular automata is the type of grid which it is computed. In one-dimensional, the grid is a one-dimensional line. In two dimensions, square, triangular, and hexagonal grids may be considered. Wolfram [32] indicated that cellular automata has many different types. The first one is the elementary cellular automata, one-dimensional cellular automata which has only two possible values for each cell (0 or 1). Another cellular automata is the universal cellular automata which is similar to the Turing machine, exhibiting universality. The last cellular automata is the totalistic cellular automata which has color, at least two colors for the rule to generate each CA illustrate again in section 2.4.3.

Wolfram [14,12] distinguished CA into four classes, based on entropy measures:

1. Evolution leads to homogeneous state. CA always reaches a state in which all cells are either dead or alive. The state can not change either, so no changes will occur after reaching this state.

2. Evolution leads to a set of separated simple stable or periodic structures. This means that every finite configuration evolves to a periodic configuration in finite number of steps.

3. Evolution leads to a chaotic pattern. For this class, everything occurs randomly. The structures that appear in these CAs' maybe ordered, but no periodic or shifting patterns can be observed.

4. Evolution leads to complex localized structures which are sometimes long-lived. This type of CA is very famous in 2D CA

2.4.2 Neighborhoods

As mentioned above neighborhood is one component of the CA computation process. As a result, many neighborhoods are introduced with their relative grid shapes. The neighborhoods for each dimension have been surveyed [27]. The Wolfram neighborhood is one of the simplest one-dimensional neighborhoods. The automata built on this neighborhood with one state per cell is called "elementary". Two popular neighborhoods in 2D CA are Von Neumann neighborhood whose name is from John Von Neumann, and another one is Moore neighborhood named after the pioneer, Edward F. Moore that is used in this research. The detail of the Moore neighborhood [28] is described below:

The Moore neighborhood is a square-shaped neighborhood. It is used to define a set of cells surrounding on four sides and four corners of a given cell. The cell may affect the evolution of two-dimensional cellular automata on a square grid. The Moore neighborhood of the range r is defined by

$$N^{M}{}_{(x0,y0)} = \{(x,y) : |x - x_0| \le r, |y - y_0| \le r\} \quad \text{(e.q 2.2)}$$

Where: N^{M} denotes the set of cells in the Moore neighborhood

x_0 denotes the x – coordinate of the current cell

y_0 denotes the y – coordinate of the current cell

x denotes the x – coordinate of another cell

y denotes the y – coordinate of another cell

r denotes the range

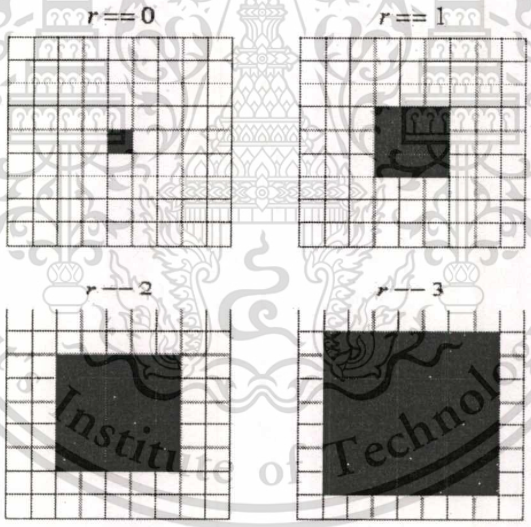


Figure 2. 5: Moore neighborhoods for ranges r = 0, 1, 2, and 3. [28]

2.4.3 Rules

The generation of rules in 1D and 2D are different. To generate one dimensional CA the rules is based on the value of the current cell, the value of the cell to its left, and the value of the cell to its right. Since there are $2^3 = 8$ possible binary states for the three cells neighboring and there are $2^8 = 256$ rules. Also in 1D totalistic cellular automata is evolved by specifying the state a given cell will have in the next generation based on the average value of the current cell and neighborhood in both sides. This research concentrated on 2D totalistic cellular automata in

which the rule generated depends only on the total or equivalent, or the average of the values of the cells in a neighborhood. Two kinds of totalistic rules for the Moore neighborhood are considered the purely totalistic rules and the outertotalistic rules, also called semi totalistic.

In the purely totalistic rules, the next state of the current cell depends only on the sum of the states of all cells in the neighborhood, including the current cell. Let k be number of colors, the 9(k-1) is the maximum possible total value of all 9 cells. There are 1 + 9(k-1) digits and $k^{1+9(k-1)}$ purely totalistic rules for 9-cell neighborhood.

In outer-totalistic rules, the next state of the current cell depends on the sum of all the cells in the neighborhood except the current cell. Let k be number of colors, the 8(k-1) is the maximum possible total value of all 8 cells neighborhood. There are k current cell states to consider, so there are k(8(k-1)+1) digits, rule tables for 8-cell neighborhood have k(8(k-1)+1) rows and $k^{k(8(k-1)+1)}$ outer-totalistic rules. The best known two-dimensional totalistic cellular automaton is the game of life [33] discovered by J. H. Conway in 1970 which is binary outer-totalistic (only two colors) for the Moore neighborhood. Each a live cell (C = 1) remains alive (C' = 1) only when surrounded by 2 or 3 live neighbors. Otherwise it dies (C' = 0). A dead cell (C = 0) comes to life (C' = 1) only when it is surrounded by exactly 3 live neighbors. The corresponding table is:

Table 2. 1: The rule table of the game of life rule

Total value of the neighborhood Σ	Value of the current cell C	Value of the current cell for next time step C'
0	0	0
0	1	0
1	0	0
1	1	0
2	0	0
2	1	1
3	0	1
3	1	1
4	0	0
4	1	0

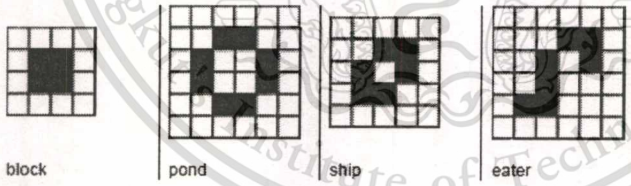
Table 2. 2: The rule table of the game of life rule (Cont)

Total value of the neighborhood Σ	Value of the current cell C	Value of the current cell for next time step C'
5	0	0
5	1	0
6	0	0
6	1	0
7	0	0
7	1	0
8	0	0
8	1	0

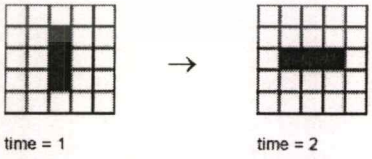
The base-2 code for this rule is 000000000011100000 which is 224 in base 10.

According to the rule, some patterns always stay the same, some behave in a periodic way, and always return to the same state, and some tend to travel around the array. Some of those are shown below.

stable patterns:



periodic pattern:



moving pattern:

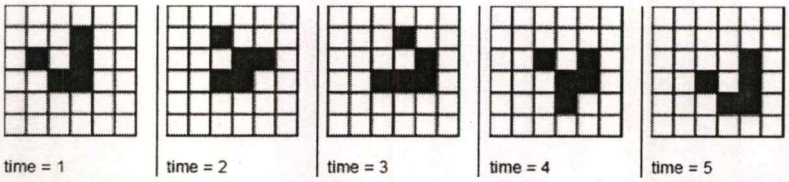


Figure 2. 6: Three characteristics of game of life rule

The fifteen cellular automata families [30] with many interesting rules are illustrated in 1D and 2D with some explanation of each rule about their characteristic and their type. This section concentrates on only 2DCA rules in Life family that are similar to the natural phenomena. An example and rule notation is explained below:

Life rules are defined in the "S/B" form, where:

S - defines counts of alive neighbors necessary for a cell to survive,

B - defines counts of alive neighbors necessary for a cell to be born.

Table 2. 3: The one example rule for the life rules

Name	Rule (S/B)	Character	Neighborhood	Description
Coral	45678/3	Exploding	Moore Neighborhood	This rule slowly generates patterns that slightly expands on every side to produces an interesting coral-like texture.

Figure 2. 7: Life family: Coral Rule [30]

2.5 OpenGL programming

Rendering in computer graphics is the process that creates images from models. These models, or objects, are constructed from geometric primitives, such as: points, lines, and polygons that are specified by their vertices. This section will briefly explain OpenGL drawing primitives, specific light and texture mapping [20].

2.5.1 Specifying vertices and geometric drawing primitives

In OpenGL, for all geometric objects using the **glVertex*()** command to specify a vertex, the command **glVertex*()** should be executed between a **glBegin()** and **glEnd()** pair. The general syntax is shown below:

void **glBegin**(GLenum *mode*);

void **glVertex**{234}{sifd}[v](*TYPEcoords*);

...

void **glEnd**(void);

The type of primitive is indicated by *mode*, which can be any of the values shown in Table 2.3.

Table 2. 4: Geometric Primitive Names and Meanings

Value	Meaning
GL_POINTS	Individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_POLYGON	Boundary of a simple, convex polygon
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUAD_STRIP	linked strip of quadrilaterals

The following figure 2.8 show the drawing geometrics primitive that correspond to the specific vertices and arguments are assigned.

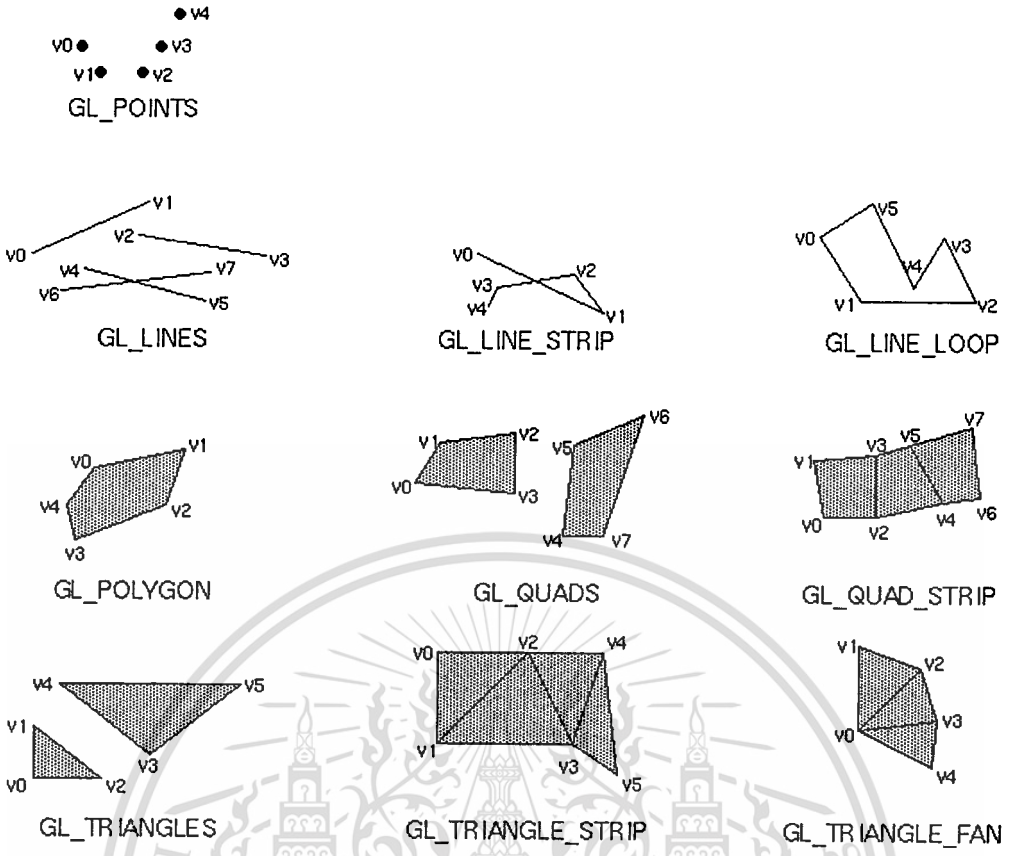


Figure 2. 8: Geometric Primitive Types

2.5.2 Lighting in OpenGL

The OpenGL lighting model considers the lighting to be divided into four independent components: emitted, ambient, diffuse, and specular. All four components are computed independently, and then added together.

The Emitted light is the simplest, it originates from an object and is unaffected by any light sources.

The ambient component is the light from the source that has been scattered so much by the environment that its direction is impossible to determine, it comes from all directions.

The diffuse light comes from one direction, so it is brighter if it comes squarely down on a surface than if it barely glances off the surface. When it hits a surface, it is scattered equally in all directions, so it appears equally bright, no matter where the eye is located.

The specular light comes from a particular direction, and it tends to bounce off the surface in a preferred direction.

2.5.2.1 Creating Light Sources

Light sources have a number of properties, such as color, position, and direction. The command used to specify all properties of lights is **glLight*()**, the syntax and meaning for each argument are shown below:

void **glLight**{if}[v](GLenum *light*, GLenum *pname*, *TYPEparam*);

Creates the light specified by *light*, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7.

The *param* argument indicates the values to which the *pname* characteristic is set, it is a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single valued light characteristics.

The characteristic of the light is defined by *pname*, which specifies a named parameter in Table 2.4.

Table 2. 5: Default Values for *pname* Parameter of glLight*()

Parameter Name	Default value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	Diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(*x*, *y*, *z*, *w*) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(*x*, *y*, *z*) direction of spotlight
GL_SPOT_EXPONENT	0.0	Spotlight exponent
GL_SPOT_CUTOFF	180.0	Spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor

Table 2. 6: Default Values for *pname* Parameter of glLight*() (Cont)

Parameter Name	Default value	Meaning
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

2.5.2.2 Defining Material Properties

This section describes how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors, the shininess, and the color of any emitted light. Most of the material properties are conceptually similar to ones that were already used to create light sources. The command to set the material properties is called glMaterial*(). The syntax and meaning for each argument are show below:

void glMaterial{if}[v](GLenum *face*, GLenum *pname*, *TYPEparam*);

The face parameter can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate which face of the object the material should be applied to. The particular material property being set is identified by *pname* and the desired values for that property are given by *param*, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version works only for setting GL_SHININESS. The possible values for *pname* are shown in Table 2.5.

Table 2. 7: Default Values for *pname* Parameter of glMaterial*()

Parameter Name	Default value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material

Table 2. 8: Default Values for *pname* Parameter of glMaterial*() (Cont)

Parameter Name	Default value	Meaning
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material
GL_COLOR_INDEXES	(0,1,1)	ambient, diffuse, and specular color indices

2.5.3 Texture mapping in OpenGL

Because realism in computer graphics needs complexity, texture mapping is the one technique that is often used for realism. Texture mapping can effectively create an appearance of complexity without modeling and rendering every 3D detail of a surface. Texture mapping means the mapping of a function onto a surface in 3D. The domain of the function can be one, two, or three-dimensional, and it can be represented by either an array or by a mathematical function. The following section will present the steps to define the texture mapping in OpenGL.

2.5.3.1 Specify the Texture

OpenGL supports 1D, 2D, and 3D textures, the command **glTexImage1D(), glTexImage2D()** and **glTexImage3D()** used to define 1D, 2D and 3D texture respectively. However, this section concentrates on 2D texture where the command **glTexImage2D()** is called. The syntax and meaning for each argument are show below:

void **glTexImage2D**(GLenum *target*, GLint *level*, GLint *components*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **pixels*);

The *target* parameter is intended for future use by OpenGL; for this release, it must be set to the constant GL_TEXTURE_2D.

To use the *level* parameter if supplying multiple resolutions of the texture map; with only one resolution, *level* should be 0.

The next parameter, *components*, is an integer from 1 to 4 indicating which of the R, G, B, and A components are selected for use in modulating or blending. A value of 1 selects the R

component, 2 selects the R and A components, 3 selects R, G, and B, and 4 selects R, B, G, and A.

The *width* and *height* parameters give the dimensions of the texture image, *border* indicates the width of the border, which is usually zero. Both *width* and *height* must have the form 2m+2b, where m is an integer and b is the value of *border*. The maximum size of a texture map depends on the implementation of OpenGL, but it must be at least 64 x 64 (or 66 x 66 with borders). If *width* or *height* is set to zero, texture mapping is effectively disabled.

The *format* and *type* parameters describe the format and data type of the texture image data. The *format* parameter can be GL_COLOR_INDEX, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, or GL_LUMINANCE_ALPHA that is, the same formats available for **glDrawPixels()** with the exceptions of GL_STENCIL_INDEX and GL_DEPTH_COMPONENT. Similarly, the *type* parameter can be GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, or GL_BITMAP.

Finally, *pixels* contain the texture image data. This data describes the texture image itself as well as its border.

2.5.3.2 Controlling Filtering

Texture maps are square or rectangular, but after being mapped to a polygon or surface and transformed into screen coordinates, the individual texels of a texture rarely correspond to individual pixels of the final screen image. Depending on the transformations used and the texture mapping applied, a single pixel on the screen can correspond to anything from a tiny portion of a texel (magnification) to a large collection of texels (minification), as shown in Figure 2.9.

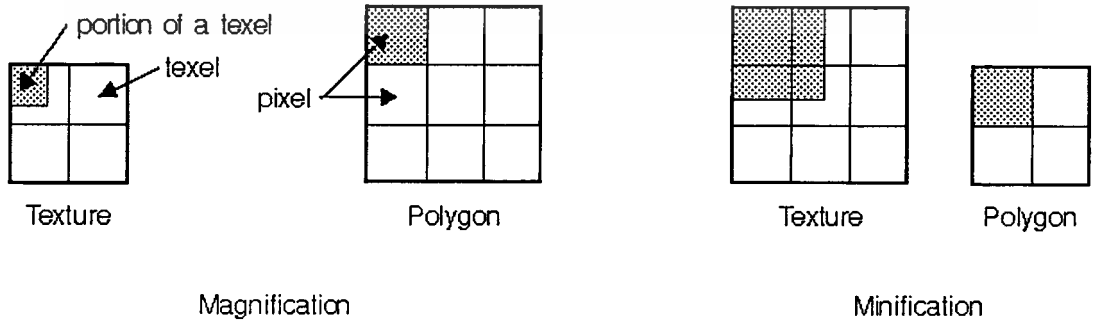


Figure 2. 9: Texture Magnification and Minification

To specify the magnification and minification filtering methods **glTexParameter*()** is used. The syntax and meaning for each argument are show below:

void glTexParameteri(GL_TEXTURE_2D, GLenum param, GLenum filter);

The first argument to **glTexParameter*()** is either GL_TEXTURE_2D or GL_TEXTURE_1D, depending on whether you are working with two- or one-dimensional textures. The second argument is either GL_TEXTURE_MAG_FILTER or GL_TEXTURE_MIN_FILTER to indicate whether it is specifying the filtering method for magnification or minification. The third argument specifies the filtering method, Table 2.6 lists the possible values.

Table 2. 9: Filtering Methods for Magnification and Minification

Parameter	Values
GL_TEXTURE_MAG_FILTER	GL_NEAREST or GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR

2.5.3.3 Modulating and Blending

The values in the texture map can use to modulate the color that the surface would be painted without texturing, or to blend the color in the texture map with the nontextured color of the surface. To set the current texturing function

The command **glTexEnv*()** is used to set the current texturing function by supplying the appropriate arguments. The general syntax and meaning of each arguments are illustrate below:

void **glTexEnv**{if}{v}(GLenum *target*, GLenum *pname*, *TYPEparam*);

target must be GL_TEXTURE_ENV

If *pname* is GL_TEXTURE_ENV_MODE, *param* can be GL_DECAL, GL_MODULATE, or GL_BLEND, to specify how texture values are to be combined with the color values of the fragment being processed.

If *pname* is GL_TEXTURE_ENV_COLOR, *param* is an array of four floating-point values representing R, G, B, and A components. These values are used only if the GL_BLEND texture function has been specified as well.

Decal mode makes sense only if the number of components is three or four. With three selected components, the color that would have been painted in the absence of any texture mapping (the fragment's color) is replaced by the texture color, and its alpha is unchanged. With four components, the fragment's color is blended with the texture color in a ratio determined by the texture alpha, and the fragment's alpha is unchanged.

For modulation, the fragment's color is modulated by the contents of the texture map. For one or two components, the color values are multiplied by the same value, so the texture map modulates between the fragment's color (if the luminance is 1) to black (if it is 0). With three or four components, each of the incoming color components is multiplied by a corresponding (possibly different) value in the texture. If there is an alpha value (which there is for two or four components), it is multiplied by the fragment's alpha.

Blending mode makes sense only for one- or two-component textures. The luminance is used somewhat like an alpha value to blend the fragment's color with the color specified by GL_TEXTURE_ENV_COLOR. With two components, the fragment's alpha is also multiplied by the alpha in the texture.

2.5.3.4 Assigning Texture Coordinates

Texture coordinates can compose of one, two, three, or four coordinates. They are usually referred to as the s, t, r, and q coordinates to distinguish them from object coordinates (x, y, z, and w) and from evaluator coordinates (u and v). The command to specify texture coordinates used **glTexCoord*()**, is similar to **glVertex*()**, it is used in between **glBegin()** and **glEnd()**. Usually, texture-coordinate values range between 0 and 1, but it can be assigned outside this range. The general syntax is shown below:

void **glBegin**(GLenum *mode*);

void **glTexCoord**{1234}{sifd}{v}(**TYPEcoords**);

void **glVertex**{234}{sifd}[v](**TYPEcoords**);

...

void **glEnd**(void);

With **glTexCoord1*()**, the **s** coordinate is set to the specified value, **t** and **r** are set to 0, and **q** is set to 1.

Using **glTexCoord2*()** allows you to specify **s** and **t**; **r** and **q** are set to 0 and 1, respectively.

With **glTexCoord3*()**, **q** is set to 1 and the other coordinates are set as specified.

One can specify all coordinates with **glTexCoord4*()**. Use the appropriate suffix (s, i, f, or d) and the corresponding value for **TYPE** (GLshort, GLint, GLfloat, or GLdouble) to specify the coordinates' data type.

The **glTexParameter()** is used to set various parameters that control how a texture is treated as it is applied to a fragment. The general syntax and meaning of each argument are show below:

void glTexParameter{if}{v}(GLenum target, GLenum pname, TYPE param);

The *target* parameter is either GL_TEXTURE_2D or GL_TEXTURE_1D to indicate a two- or one-dimensional texture. The possible values for *pname* and *param* are shown in Table 2.8. You can use the vector version of the command to supply an array of values for GL_TEXTURE_BORDER_COLOR, or you can supply them individually using the nonvector version. If these values are supplied as integers, they're converted to floating-point, they're also clamped to the range [0,1]

Table 2. 10: Arguments for glTexParameter*()

Parameters	Values
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR

Table 2. 11: Arguments for glTexParameter*() (Cont)

Parameters	Values
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	any four values in [0, 1]

2.5.3.5 Automatic Texture Coordinate Generation

Actually, the mechanism for automatic texture-coordinate generation is useful for many different applications such as showing contours and environment mapping. To generate texture coordinates automatically, use the command **glTexGen()**.

void **glTexGen**{ifd}{v}(GLenum **coord**, GLenum **pname**, **TYPEparam**);

The first parameter, *coord*, must be GL_S, GL_T, GL_R, or GL_Q to indicate whether texture coordinate *s, t, r*, or *q* is to be generated.

The *pname* parameter is GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If it's GL_TEXTURE_GEN_MODE,

param is an integer (or, in the vector version of the command, points to an integer) that's either GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. These symbolic constants determine which function is used to generate the texture coordinate.

- **Creating Contours**

When GL_TEXTURE_GEN_MODE and GL_OBJECT_LINEAR are specified, the generation function is a linear combination of the object coordinates of the vertex (xo,yo,zo,wo):

generated coordinate = $p1x0 + p2y0 + p3z0 + p4w0$

The $p1$, ..., $p4$ values are supplied as the *param* argument to **glTexGen*()** with *pname* set to GL_OBJECT_PLANE. With $p1$, ..., $p4$ correctly normalized, this function gives the distance from the vertex to a plane.

2.6 3 Dimensional Surface Cellular Automata Method

The 3 dimensional Surface Cellular Automata (3DSCA) method is introduced by Grobon and Chiba [16] in 1999 with two applications: 3D surface Voronoi Diagram, and Corrosion and Patina Generation. The purpose of this method used is to generate and render any 3-dimentional model. The objects in this method are in the form of triangle mesh. It means that any 3D object is divided into triangles. The generation of its cellular automata rules are based on the actual shaped of any triangle mesh which gives the direct texturing simulation. This was the first 3D cellular automata texturing method with specific domain. A few years later, they continued developing their method to simulate Crack Pattern [17] and Peeling [18]. In recent years, Philippe Even and Stephane Gobron presented a method to simulate some natural phenomena on a building by applied 3DSCA [13]. The details of the 3DSCA method are explained below:

2.6.1 3DSCA Model

The model of this method can be applied with any computer graphics applications. This model consists of one or many objects with dependent cellular automata (CA). As mentioned above, each object is subdivided into triangles and each triangle is subdivided into a grid of identical cells. The projection of three vertices *A, B*, and *C* has to be in such a way that *A* is always *(0.0, 0.0), B (0.0, By),* and *C (Cx, Cy)* with *Cy >= 0.0*. Fig. 2.10 also shows theses points a simple cube object.

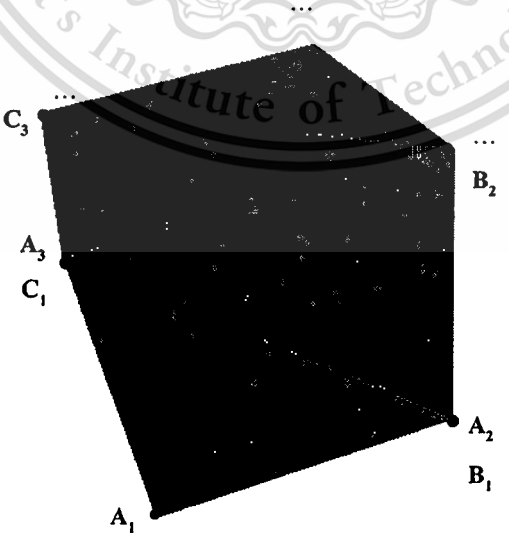


Figure 2. 10: A cube, an example the 3DSCA model

The geometry of a cell is considered in three cases according to their position and the triangle ABC. Fist case, some geometry of the cells is inside triangle, in another case the geometry of the cells is partially in the triangle, and in the last case the geometry of the cells is completely out. Thus, the hypotheses for defining a cell in the model are considered:

The cell has only 8 neighbourhoods surrounded by four sides and four corners.

A cell must have an in/out flag, the cell is considered as active in the CA when it is at least partially in the input triangle.

The cells are completely out of the triangle ("⊗"symbol in Figure 2.11). They must be considered as none existent and define the in/out flag as being *Cell_Out.*

Two very useful in/out flag positions are the *Cell_Part* and *Cell_In* positions. If the center point of the cell ("○") is outside the triangle, then the cell is *Cell_Part* position. If the center point of the cell ("●") is inside the triangle, then the cell is *Cell_Part* position. These two states are only optional because they do not interact with the CA behavior. However, they are very useful in the rendering step. When the cell is active, it's *IN flag* position is determined by the geometric center of the cell.

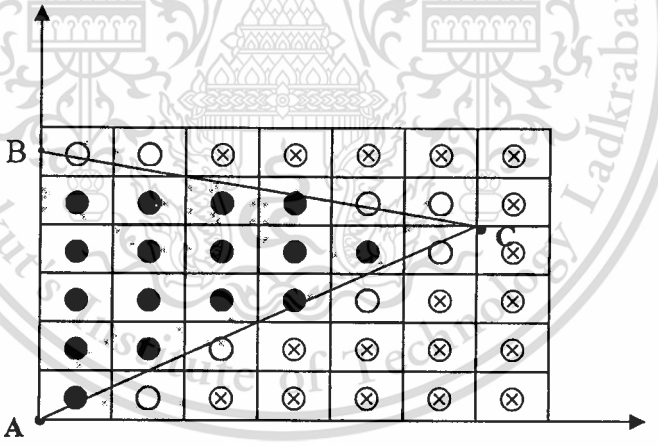


Figure 2. 11: The project triangles.

2.6.1.1 Rendering Using OpenGL

Rendering technique is an important technique in computer graphics. The 3DSCA method try to keep the render as general as possible, in order to preserve greater flexibility. So to support this request, the 3DSCA has its own light model. To insure that the display information of each cell in the CA must not be lost, this method uses Triangle Strips (TS) and Triangle Fans (TF) in the OpenGL program [9] for rendering structures. Both of them provide very fast and convenient

rendering. The following paragraphs describe simple light model and explain how to use Triangle Strips (TS) and Triangle Fans (TF).

Simple Light Model

We have: $C_{[RGB]} = \left(K_a + K_d . \vec{L}.\vec{N}_c + K_s . \left(\vec{R}.\vec{E}\right)^n\right) C_{m[RGB]}$ (e.q 2.3)

Where: K_a denotes: The ambient term, the value is a constant.

K_d denotes: The diffuse term

$\vec{L}$ denotes the light vector

$\vec{N}_c$ denotes the cellular normal vector.

K_s denotes the specular term follows Phong's specular model [3]

$\vec{R}$ denotes: The light reflection vector

$\vec{E}$ denotes the camera (or 'eye') orientation

n denotes the specular power of the material.

C_m denotes the initial material color.

- **Triangle Strips and Triangle Fans**

This section shows how to use OpenGL Triangle Strips (TS) and Triangle Fans (TF) as data structures for rendering. Each triangle has be reconstructed with a set of triangle bands (TB = TS or/and TF), so that TB vertices correspond as much as possible to the cell center. In order to achieve this, there are two main steps to consider:

(a) The contour of the triangle and three series of intermediate points are defined by the three vertices *A*, *B* and *C*. The intermediate points are achieved by the intersection of the centered cell grid line and the three triangle edges. As the corresponding cells of these points are only partially in the triangle, their colorings are computed with a convolution method.

(b) Dividing each triangle into three regions (Figure 2.12):

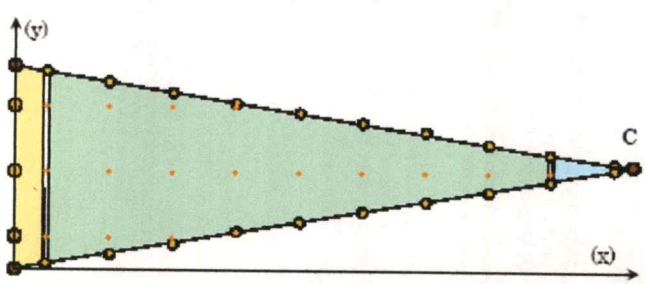


Figure 2. 12: Three regions (yellow, green, and blue) subdividing input triangles

: The first band is always defined as TB and its size is ½ grid unit;

: The intermediate region contains a set of similar TB (identical thickness);

: This region depends on the geometric position of the point *C*. If *Cx* is smaller or equal to the half cell, then this region is only one TF, otherwise it has to be decomposed into one TB and one TF.

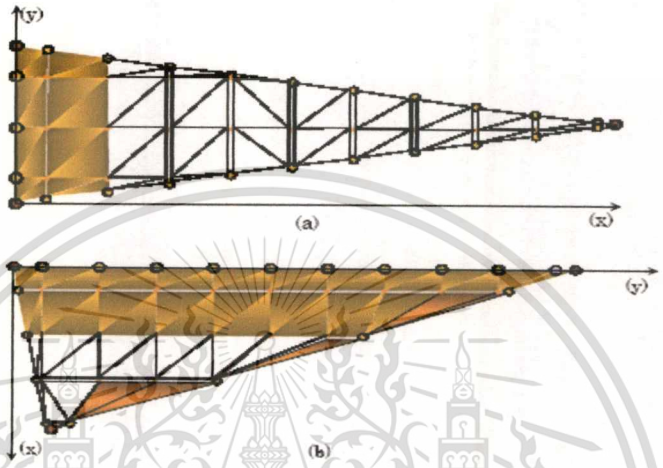


Figure 2. 13: Two possible types of triangle to recompose [16]

CHAPTER 3

THEORY

As mentioned in chapter1, two techniques are used in this research to simulate the weathered wood, the texture mapping technique and the 3DCSA method. The hearts of this research is the wood texture generated by the Perlin Noise technique and the rule of Cellular Automata. This chapter will present the technique on how to generate wood texture, and will present the new rules of Cellular Automata that correspond to the change of color of the wooden objects. Finally, the technique used to construct the 3DSCA model is exhibited.

3.1 Generating Wood Texture

This section describes all steps of the method to generate wood textures. The color of wood has to be considered first. Actually, every color comes from three primary colors, namely red, green and blue. Also in the real 2D image, the value of each color is between 0 and 255, the darkness and brightness of the color depends on its value in the range. The color is very dark if its value is 0, and very bright if its value is 255. The mixtures of three colors become to others colors. The brightness or darkness depends on the amounts of the three mixed colors.

This research concentrates only on 2D textures where the technique creates the wood textures developed from [31]. The steps of creating wood texture are: First create three following functions, a Noise Function, a SmoothNoise Function and a Turbulence Function. Finally, the function for creating wood texture is created. The following paragraphs will show the functions used and explain how they work.

The Noise function is a function in which random real values belonging to a formula are assigned. For the formula used to create random noise we can use any equation that gives a suitable value. The result of this function is the random real values between -1.0 and 1.0. The source code of the noise function is illustrated below:

```
float noise(int x,int y)
{
        int n = x + y*57;
        n = (n<<13)^n;
return (1.0-((n*(n*n*15731+789221)+1376312589)& 0xffffffff)/1073741824.0);
}
```

Figure 3. 1: The Noise function source

Figure 3.2 shows the value of three colors and the result after applied in the noise function:

R220 G 140 B 80

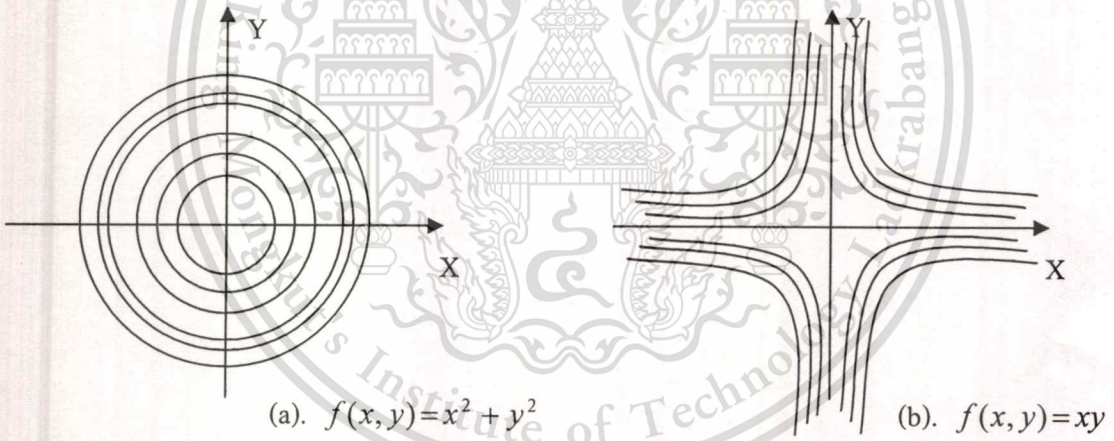
Figure 3. 2: The texture is generated by Noise function

As mentioned before, the Noise Function can be used to randomise the noise to create unpredictable noises patterns. The SmoothNoise function is used to re-generate the output of the noise function again, to make the 2D texture smoother, less random looking, and with fewer squares. The source code for creating SmoothNoise function is shown below:

```
float SmoothNoise(int x, int y)
{
  float corners = (noise(x-1,y-1) + noise(x+1,y-1) + noise(x-1,y+1) + noise(x+1,y+1))/16;
  float  sides = (noise(x-1,y) + noise(x+1,y) + noise(x,y-1) + noise(x,y+1))/8;
  float center = noise(x,y)/4;
  return corners + sides + center;
}
```

Figure 3. 3: The SmoothNoise function source code

For getting a beautiful pattern, the basic graphs of mathematical functions are expected. The mathematical graphs that have patterns similar to wood patterns, are the aim of this section. Some equations of the functions that can create graphs which are similar to the wood patterns and its corresponding textures are shown below:



(a). $f(x,y)=x^2+y^2$ (b). $f(x,y)=xy$

Figure 3. 4: The basic graphs of the function *f(x,y)*

After apply the function $f(x,y)=x^2+y^2$, we can generate a texture like this:

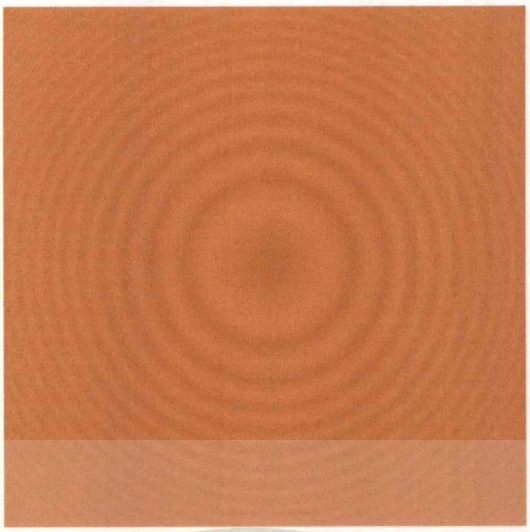


Figure 3. 5: The texture is generated base on $f(x, y) = x^2 + y^2$

After apply the function $f(x, y) = xy$, we can generate a texture like this:

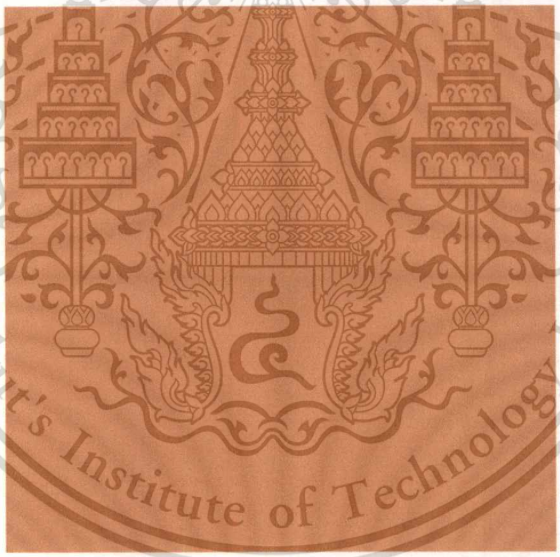


Figure 3. 6: The texture is generated based on $f(x, y) = xy$

Another texture is generated based on the function $f(x, y) = y^2$. It shows another form of wood pattern:

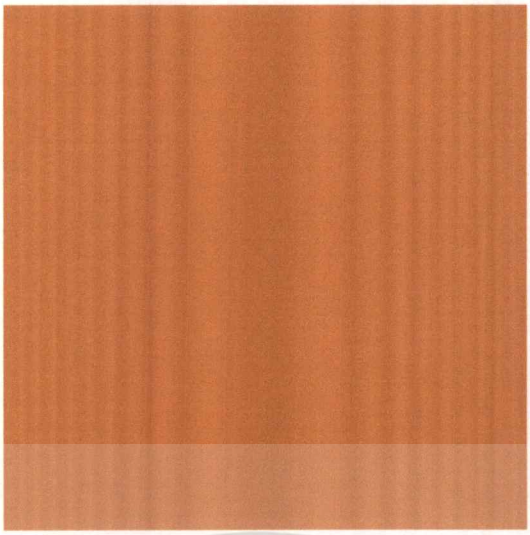


Figure 3. 7: The texture is generated based on $f(x, y) = y^2$

One more texture is generated based on the function $f(x,y)=x^2/a^2+y^2/b^2$. Its wood pattern is shown below:

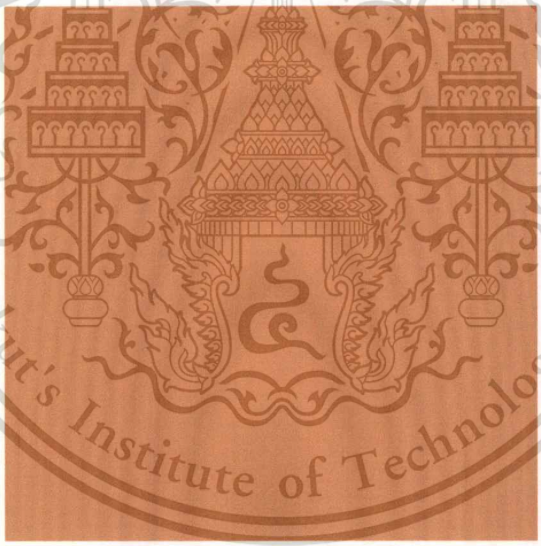


Figure 3. 8: The texture is generated based on $f(x,y)=x^2/a^2+y^2/b^2$, *where* $y \geq 0$.

However, the textures do not look like realistic wood patterns because their patterns are not complicated. As a result, to create natural realistic wood textures, the Turbulence function has to be created. The Turbulence function creates natural looking features out of Smoothed noise function. The scaling down in size of the smooth noise texture is accumulated to get the Turbulence function. The scaling down started at initial size, and is divided by two each time. Keep doing this until the zooming factor is 1. The return value is normalized so that it will be a number between 0 and 255. The source code for creating Turbulence function is shown below:

```
float Turbulence(float x, float y, float size)
{
        float value = 0.0;
        float initialSize = size;
        while (size >= 1)
        {
                value += smoothNoise(x/size,y/size)*size;
                size /= 2.0;
        }
        return (128.0*value/initialSize);
}
```

Figure 3. 9: The Turbulence function source code

The last function to complete creating a realistic wood pattern is the function that takes the basic pattern of the wood from any *f(x,y)*. After the function *f(x,y)* is applied, the output texture will show the pattern based on the basic graph of *f(x,y)*, continuously, adding the Turbulence function to the function *f(x,y)* for generating the natural pattern of wood. The pattern is distorted by assigning the value of Power of Turbulence, the value of Turbulence Size and the value of the *x, y* period. The change of these three values can generate different patterns.

The Source code for the last function, and the two textures with the same function, but different pattern are shown in Figure 3.10:

```
void createTexture()
{
GLubyte myTexture[textureWidth][textureHight][3];
 float xyPeriod;
 float xValue,yValue,distValue,turbPower,turbSize;
 float sinevalue;
 xyPeriod = 20; // change to max for more line
 turbPower = 2;
 turbSize = 128.0;
glEnable(GL_TEXTURE_2D);
for(int u=0;u<textureWidth;u++)
            for(int v=0;v<textureHight;v++)
            {
             xValue = (u)/(float)(textureHight); //Half textureHight
             yValue = (v-textureWidth/2)/(float)(textureWidth);
distValue = (xValue*xValue/9 + yValue*yValue)+ turbPower*turb(u,v,turbSize)/256;
             sinevalue = fabs(sin(2*xyPeriod*distValue*3.1415));
             myTexture[u][v][0]=(190+15*sinevalue);
             myTexture[u][v][1]=(120+15*sinevalue);
             myTexture[u][v][2]=(70+10*sinevalue);
            }
glGenTextures(1,&blueTexture);
glBindTexture(GL_TEXTURE_2D, blueTexture);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR_MIPMAP_LINEAR)
;
gluBuild2DMipmaps(GL_TEXTURE_2D,3,textureWidth,textureHight,GL_RGB,GL_UNSIGNED_BYTE,
myTexture);
}
```

Figure 3. 10: The last function source code for generating the wood texture

The figure 3.11.a and the figure 3.11.b show the patterns of wood that are generated by the same function $f(x,y)=x^2/a^2+y^2/b^2, y\geq 0$, but determination of the values of xy Period variable, Turbulence Power variable and Turbulence Size variable in two patterns are different.

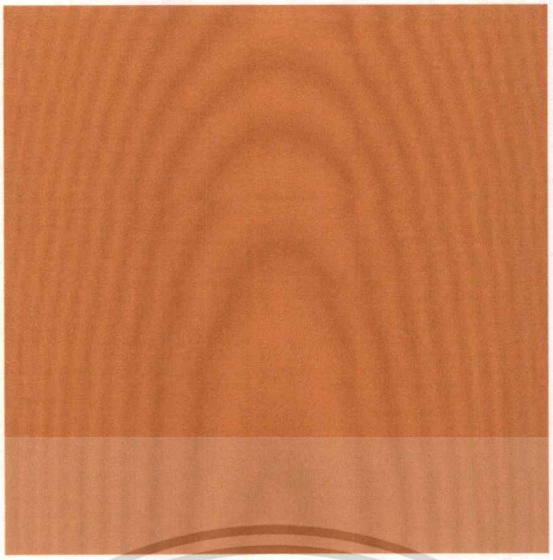


Figure 3.11.a : $f(x,y)=x^2/a^2+y^2/b^2, y\geq 0$; xy Period = 20, Turbulence Power = 1, turbulence Size = 128.0

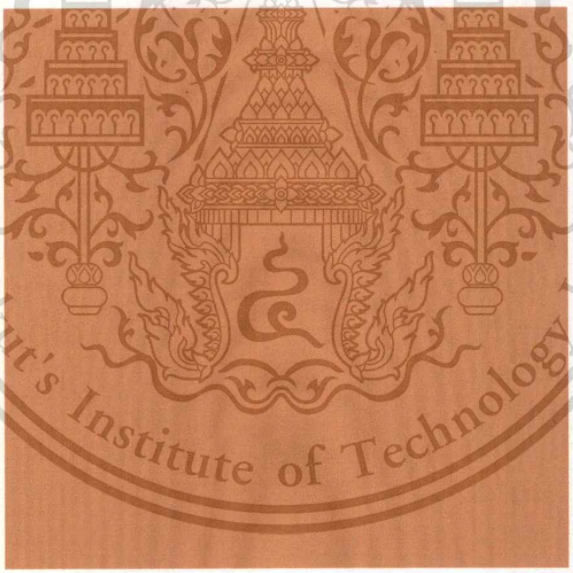


Figure 3.11.b: $f(x)=x^2/a^2+y^2/b^2, y\geq 0$; xy Period = 30, Turbulence Power = 2, Turbulence Size = 128.0

3.2 Creating the new rules of CA

According to the changes in color of wood in nature over time, the color of the wood slightly changes from bright to darker, and slowly changes from dark to silver gray color that is called weathered wood. The simulation of the change of the color of the wood in this research

defined the rule of CA in the outer totalistic cellular automata. As mentioned in section 2.4.3, the cell's color in this rule consists of 8 surrounding neighborhoods. Each cell's color has two possible values, namely: the value in dead status and the value in a live status. The generation of each current cell's color depends on the sum value of its neighborhood. The two possible rules of CA are exhibited for generating the change of the color of the wood namely: rule 25/357 and rule 25/2356. They are defined in the "S/B" form, where S denotes the sum value of neighborhood that determined the cell's status to survive and B denotes the sum value of neighborhood that determined the cell's status to be alive for the cell in dead status. For example, the rule 25/357 means that if the sum value of neighborhood is equal to 2 or 5, the current cell remains the same status. If the sum value of neighborhood is equal to 2 or 3 or 5 or 6, and the status of the current cell is dead, its status will be changed to a lived status. The table rule and the general pattern for each rule are presented in table 3.1, table 3.2 respectively.

Table 3.1: The rule table of rule 25/357

Total value of the neighborhood	Value of the current cell	Value of the current cell for next time step
0	0	0
0	1	0
1	0	0
1	1	0
2	0	0
2	1	1
3	0	1
3	1	1
4	0	0
4	1	0
5	0	1
5	1	1
6	0	0
6	1	0
7	0	1

Table 3.1: The rule table of rule 25/357 (Cont)

Total value of the neighborhood	Value of the current cell	Value of the current cell for next time step
7	1	1
8	0	0
8	1	0

Table 3.2: The rule table of rule 25/2356

Total value of the neighborhood	Value of the current cell	Value of the current cell for next time step
0	0	0
0	1	0
1	0	0
1	1	0
2	0	1
2	1	1
3	0	1
3	1	1
4	0	0
4	1	0
5	0	1
5	1	1
6	0	1
6	1	1
7	0	0
7	1	0
8	0	0
8	1	0

The general pattern for the rule 25/357 and the rule 25/2356 are presented in figure 3.13 and figure 3.14. N is the sum values of the neighborhood, C is the value of the current cell and C' the value of the current cell in the next time step.

- **Rule 25/357**

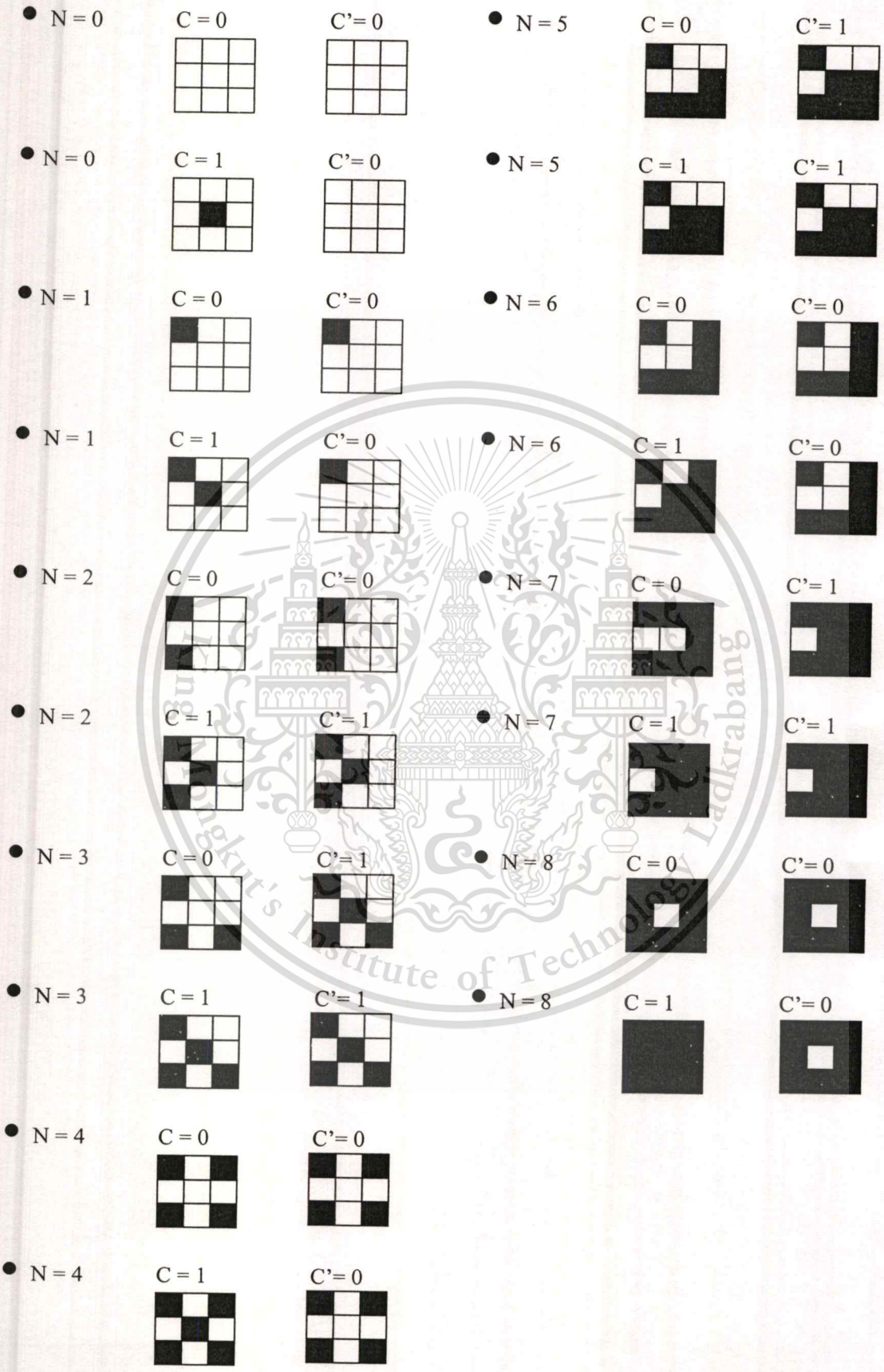


Figure 3.12: General pattern of the rule 25/357

● **Rule 25/2356**

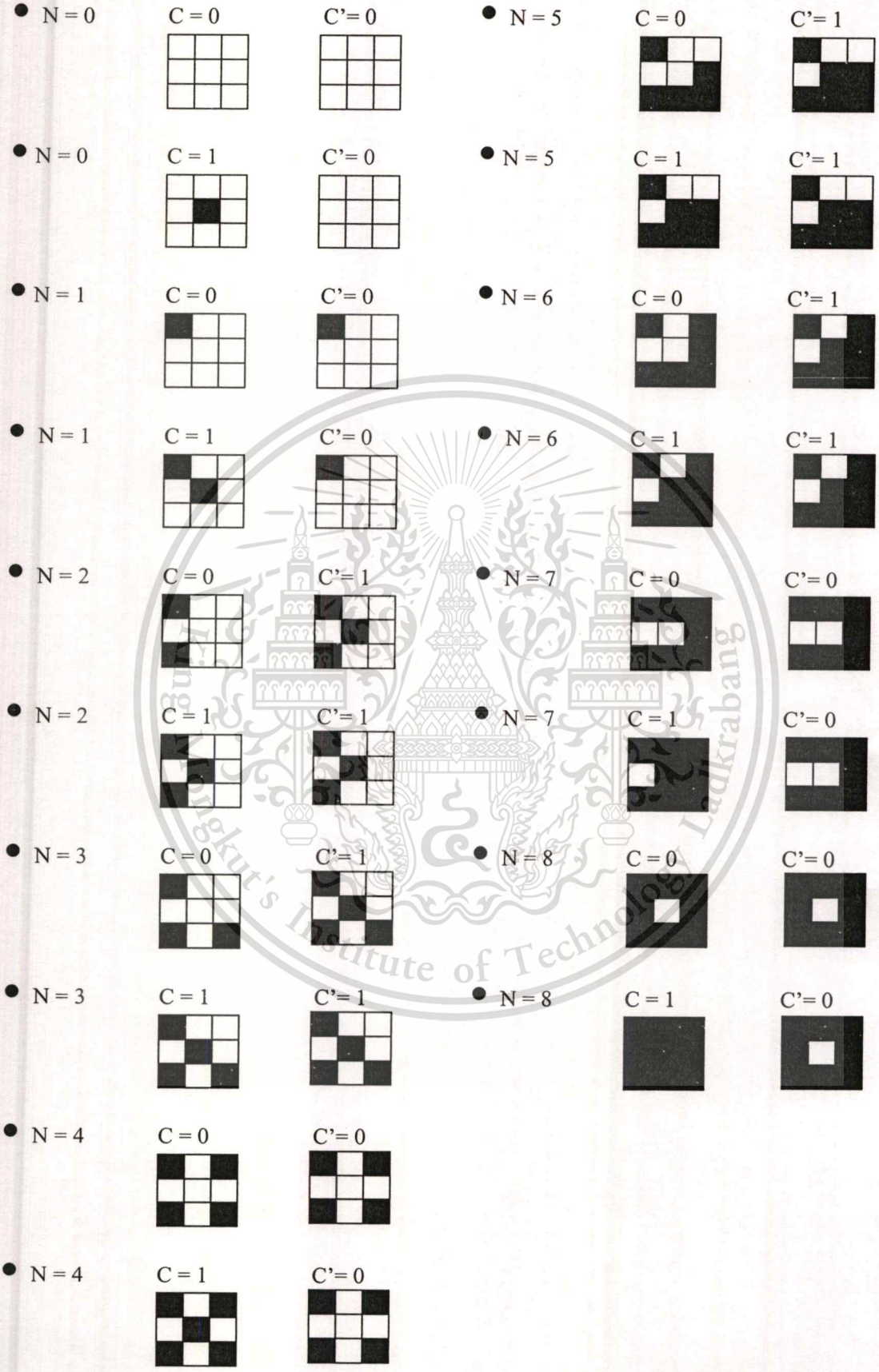


Figure 3.13: General pattern of the rule 25/2356

3.3 Construction of the 3D wooden objects

3D wooden objects are important components of this research. To create 3D wooden objects, many 3D computer graphics application programs can be used, such as: Maya, 3Ds Max, etc. As a result, creating 3D wooden objects is a simple step in this research. However, the application program for creating 3D wooden objects for this research must be able to export 3D wooden object functions into the program that we use to generate effects on wood.

The 3D models that were chosen to be the 3D wooden objects in this research are separated into two groups depending on the source. First group is the 3D models that have all parts of them directly created in side 3Ds max program. Second group is the 3D models that are not directly created by 3Ds max program, but are imported into the 3Ds max program from outside. The following figures show the examples of the 3D models from different sources. The figure 3.14 is the 3D model that was directly created by 3Ds max and visualized from the photograph in Figure 3.15. In contrast, the Figure 3.16 is the 3D model that is imported from outside.

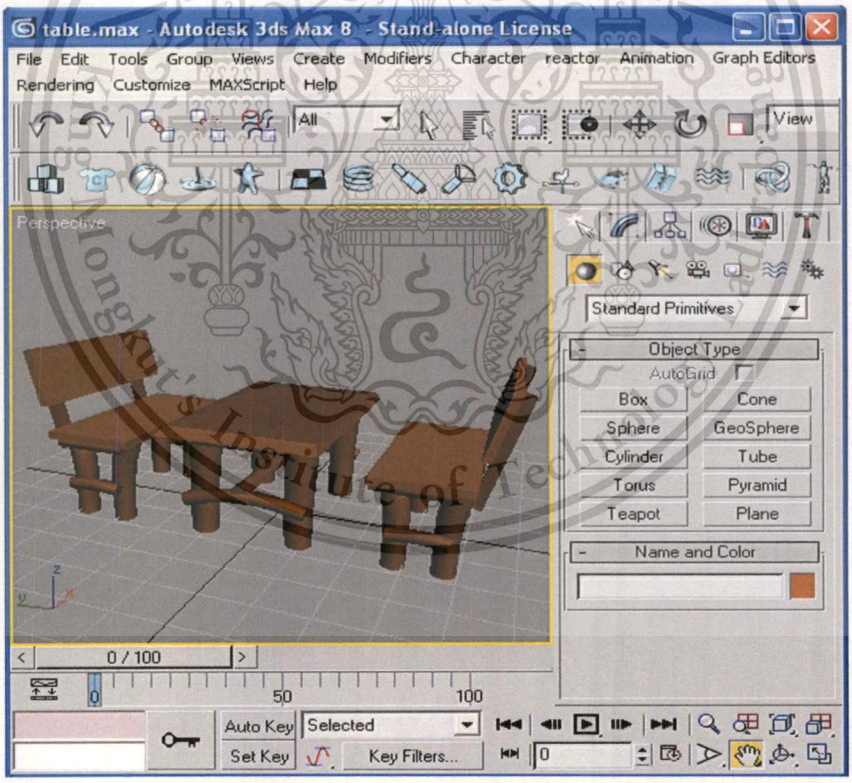


Figure 3.14: The table model visualized from Figure 3.16, creating it directly by 3Ds max program

Figure 3.15: The photograph of a table

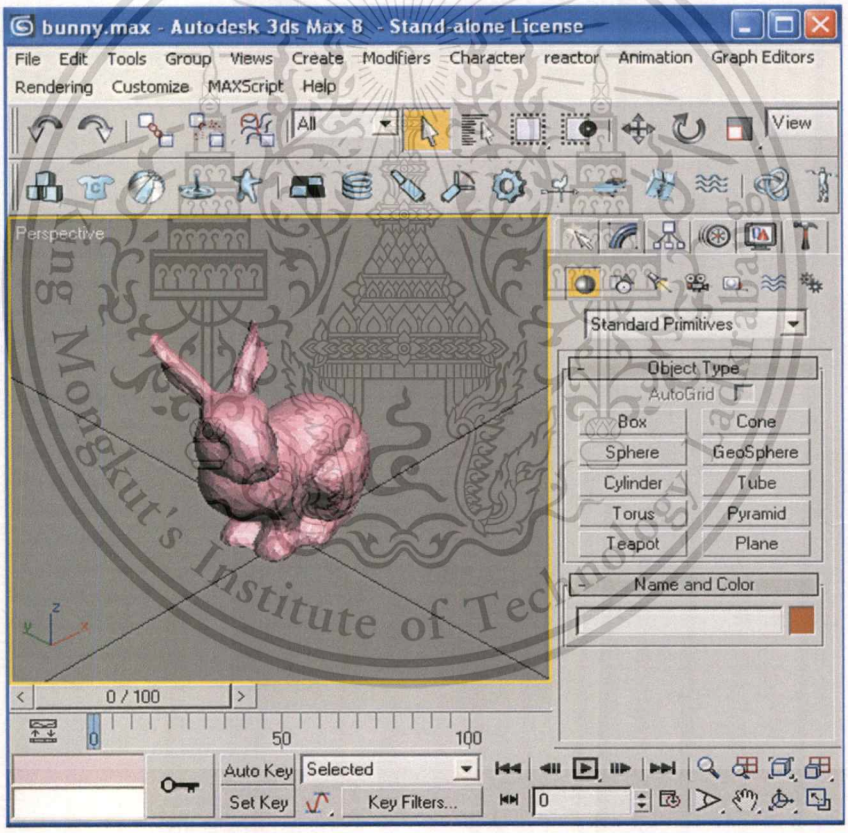


Figure 3.16: The Stanford Bunny model that is imported by 3Ds max.

Actually, when creating a 3D object in the 3Ds max, the structure of the object is assigned automatically and its component values such as: the amount of the object segments and the size are also shown on the right side of the 3Ds max window. The structure of the 3D model has to be considered in the constructing step, because the structure of the 3D model is an important factor

in rendering the 3D object. If the amount of object's segments is too small, it causes the triangle meshing of the 3D model to be slim, and the ratio of the grids and the size of the triangles not to be balanced. In contrast, if the amount of the object segments is too large, it leads to too many unexpected triangles, and the executing and rendering of CA rule to be too slow. Nevertheless, the change of the structure of the 3D model that is imported from outside depends on the support of its extension file format.

Figure 3.17.a and Figure 3.17.b show some Cylinders with two types of the segmentations.

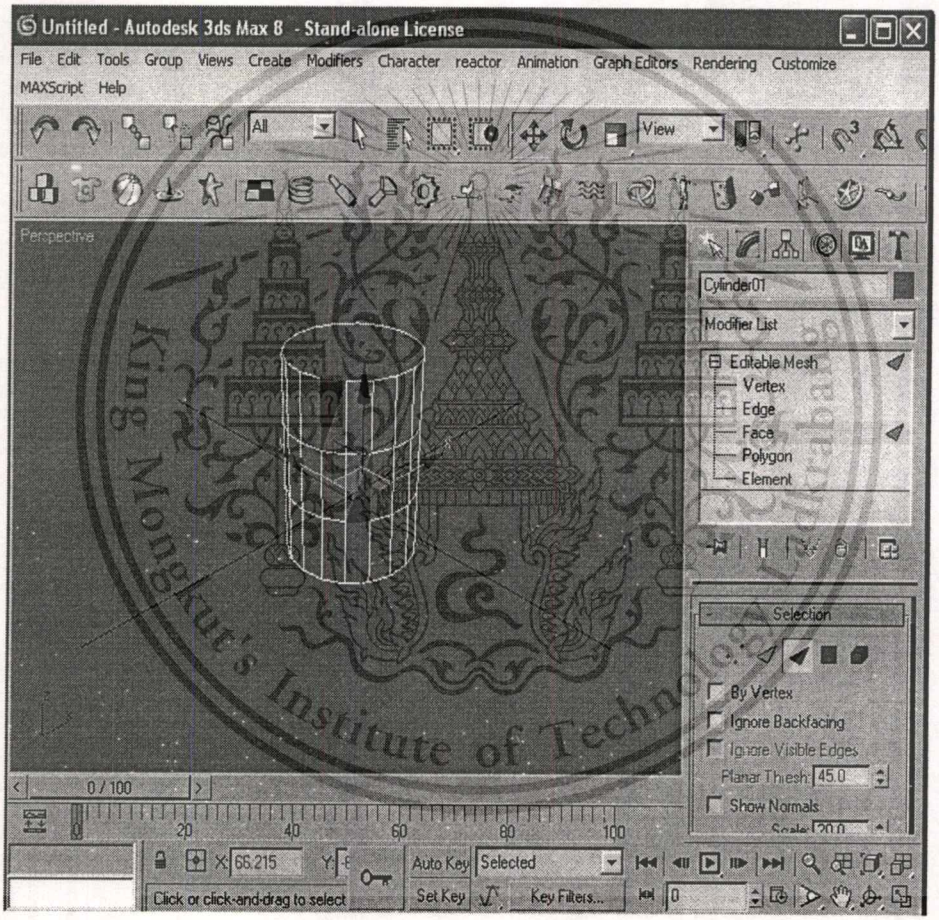


Figure 3.17.a The Cylinder model with its slim triangle meshes

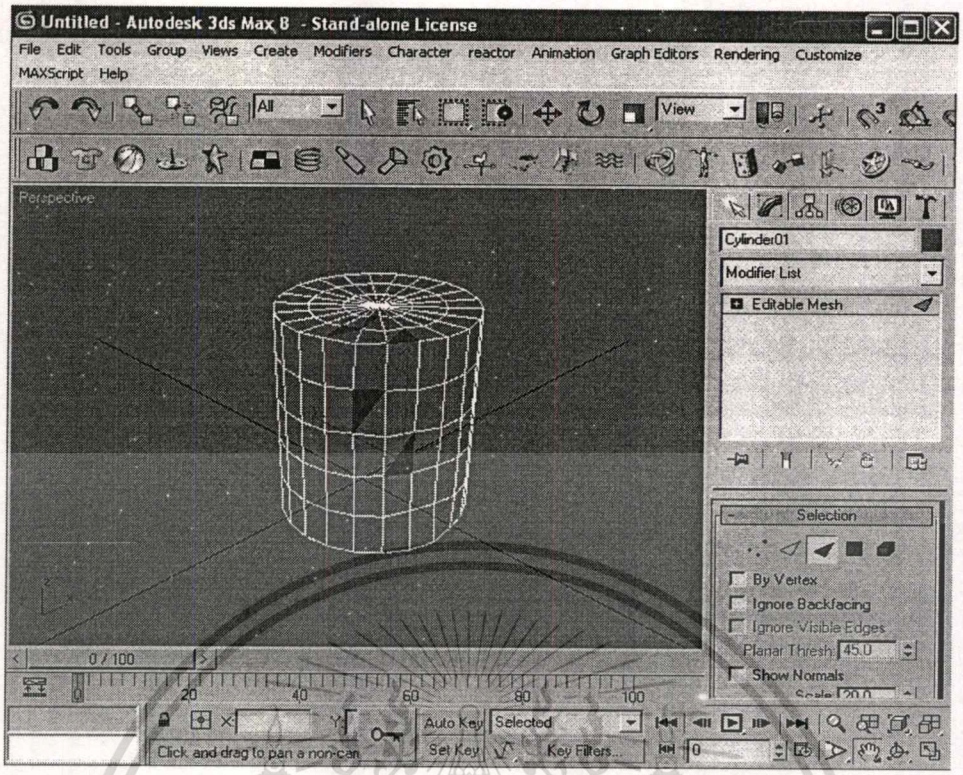


Figure 3.17.b The Cylinder model with too unexpected triangle meshes

The last step in constructing the 3D model is to export the 3D model into .ase extension file where this extension file is an ASCII text based format associated with 3DS Max. Normally, the ASE file contains only basics properties of the model, namely: coordinates and Vertices of the model.

3.4 Reconstructing the 3D Model in form of the 3DSCA model

The 3D model is reconstructed in the form of 3DSCA model. It means that the new 3D model is in the form of triangle mesh. To achieve a 3D model, every one of the three vertices of the 3D model are loaded, and denoted A, B and C respectively. According to the 3DSCA method, every 3D model has to contain all characteristics of the 3DSCA method, such as: cell properties and rules to generate a cell. The construction processes can be presented in the following flowchart and each process is illustrated in following section respectively.

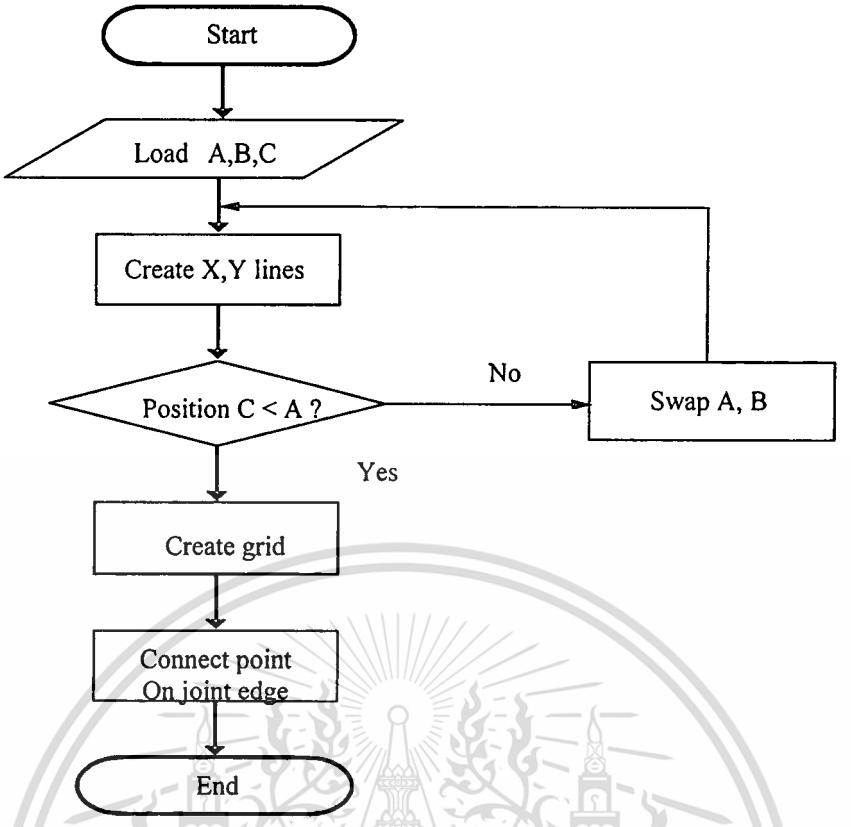


Figure 3.18: Flow chart of the reconstruction of the 3D model processes.

3.4.1: Creating grid

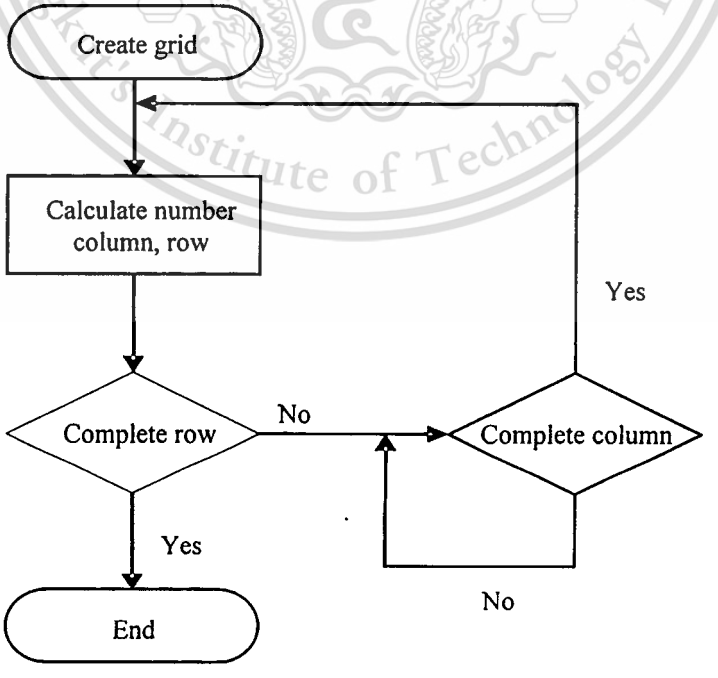


Figure 3.19: Flow chart of the process to creating the grid point

To create the cells grid to cover the triangle, we have to start with the AB line ($\overrightarrow{AB}$). It means that the vertical line of grid is parallel to $\overrightarrow{AB}$ and the horizontal line of grid is parallel to the line that passes A point and is normal to $\overrightarrow{AB}$. Look at Figure 3.21 for more details. The following paragraphs describe how to create the grid in term of mathematical methods.

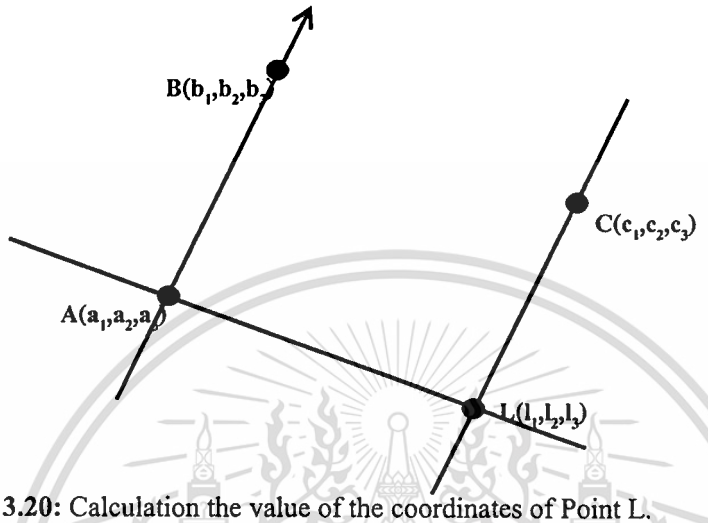


Figure 3.20: Calculation the value of the coordinates of Point L.

From Figure 3.20 to calculate the value of the coordinate of the point L, first, we assume that point L is anywhere, but line AB is parallel to line CL ($\overrightarrow{AB} // \overrightarrow{CL}$) and line AL is normal to line AB ($\overrightarrow{AL} \perp \overrightarrow{AB}$). Let $\overrightarrow{AB}$ be a direction vector of $\overrightarrow{AB}$, vector CL ($\overrightarrow{CL}$) be a direction vector of $\overrightarrow{CL}$ and vector AL ($\overrightarrow{AL}$) be a direction vector of $\overrightarrow{AL}$. According to the parallel properties and the normal properties of two lines we can calculate the value of coordinates of the point L. The statements of the calculation in mathematical solution are illustrated below:

From point $A\ (a_1, a_2, a_3)$ and point $B\ (b_1, b_2, b_3)$, the coordinate of $\overrightarrow{AB}$ is:

$$\overrightarrow{AB} = (b_1 - a_1, b_2 - a_2, b_3 - a_3) = (X, Y, Z) \qquad \text{(e.q 3.1)}$$

From point $C(c_1, c_2, c_3)$ and point $L(l_1, l_2, l_3)$, the coordinate of $\overrightarrow{CL}$ is:

$$\overrightarrow{CL} = (l_1 - c_1, l_2 - c_2, l_3 - c_3) \qquad \text{(e.q 3.2)}$$

From point $A\ (a_1, a_2, a_3)$ and point $L(l_1, l_2, l_3)$, the coordinate of $\overrightarrow{AL}$ is:

$$\overrightarrow{AL} = (l_1 - a_1, l_2 - a_2, l_3 - a_3) \qquad \text{(e.q 3.3)}$$

The equation of $\overrightarrow{CL}$ created from points $C(c_1,c_2,c_3)$ and direction vector $\overrightarrow{A\ B}=(X,Y,Z)$ is:

$$\frac{x-c_1}{X}=\frac{y-c_2}{Y}=\frac{z-c_3}{Z} \qquad \text{(e.q 3.4)}$$

From $\overrightarrow{AB}\perp\overrightarrow{AL}$, so $\overrightarrow{AB}.\overrightarrow{AL}=0$ (e.q 3.5)

The subtraction between (e.q 3.4) and (e.q 3.5), finally the value of the coordinates of point L are achieved.

In the three dimensional system, the point A, B, C and L are in the same plane. The position of the point C is an important point for creating grids that cover the ABC triangle (ΔABC). If the position of the point C locates below the point A, it causes some areas of the triangle not to be able to create grids. Thus, one more condition has to be checked, is that $\overrightarrow{LC}$ and $\overrightarrow{AB}$ are the same direction or not (using (e.q 3.8) or (e.q 3.9)). If they are not the same direction, it means that the position of the point C locates below the point A, then the position of point A and point B are swapped. The equations used to test direction of vectors $\overrightarrow{LC}$ with coordinates (x_1,y_1,z_1) and $\overrightarrow{AB}$ with coordinates (x_2,y_2,z_2) are illustrated below:

Let $\overrightarrow{U}_{LC}$ be a unit vector of $\overrightarrow{LC}$ and has the same direction as $\overrightarrow{LC}$ and is calculated by (e.q 3.6).

$$\overrightarrow{U_{LC}}=\left(\frac{x_1}{\sqrt{x_1^{\,2}+y_1^{\,2}+z_1^{\,2}}},\frac{y_1}{\sqrt{x_1^{\,2}+y_1^{\,2}+z_1^{\,2}}},\frac{z_1}{\sqrt{x_1^{\,2}+y_1^{\,2}+z_1^{\,2}}}\right) \text{(e.q 3.6)}$$

Let $\overrightarrow{U_{AB}}$ be a unit vector of $\overrightarrow{AB}$ and has the same direction as $\overrightarrow{AB}$ and is calculated by (e.q 3.7).

$$\overrightarrow{U_{AB}}=\left(\frac{x_2}{\sqrt{x_2^{\,2}+y_2^{\,2}+z_2^{\,2}}},\frac{y_2}{\sqrt{x_2^{\,2}+y_2^{\,2}+z_2^{\,2}}},\frac{z_2}{\sqrt{x_2^{\,2}+y_2^{\,2}+z_2^{\,2}}}\right) \text{(e.q 3.7)}$$

If $\overrightarrow{U_{LC}}=\overrightarrow{U_{AB}} \Rightarrow \overrightarrow{LC}$ *and* $\overrightarrow{AB}$ are the same direction (e.q 3.8)

If $\overrightarrow{U_{LC}}=-\overrightarrow{U_{AB}} \Rightarrow \overrightarrow{LC}$ *and* $\overrightarrow{AB}$ are reverse direction (e.q 3.9)

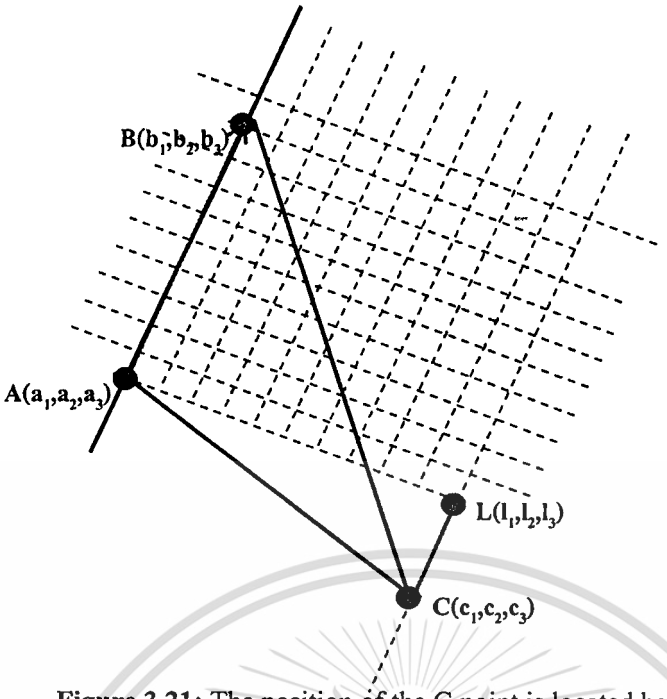


Figure 3.21: The position of the C point is located below the point A

For more detail, the explanation of creating the grid is shown Figure 3.22. The horizontal and vertical points have to be determined from the following steps. First, the set of vertical points belonging to $\overrightarrow{AB}$ are created. The distance from the point A to the first vertical point equals to the grid height, and the distance of the next respective vertical point equals to the result of the previous distance plus the grid height. These vertical points are achieved by using equation (3.8). To determine the last vertical point (P point in Figure 3.22), select the longer length between the length of $\overrightarrow{AB}$ ($\left|\overrightarrow{AB}\right|$) and $\overrightarrow{LC}$ ($\left|\overrightarrow{LC}\right|$), it is then divided by the length of grid cell. If the remain of the division is not zero, the last vertical point (point P in Figure 3.22) is located above the point B, otherwise it is located in the point B. To define the horizontal points is the next step, these points are along to the line AL ($\overrightarrow{AL}$). The point R is the last point. The calculation of the horizontal points is similar to the calculation of the vertical points. To create the grid points to cover the ΔABC, the new point Q (in Figure 3.22) is created by the corresponded two conditions $\overrightarrow{PQ} // \overrightarrow{AR}$ *and* $\overrightarrow{QR} \perp \overrightarrow{AR}$ and then the points which belong to $\overrightarrow{PQ}$ *and* $\overrightarrow{RQ}$ are created. Now we have already the points along the four sides of the rectangle APQR, after that the lines that are parallel to the line AR ($\overrightarrow{AR}$) and the line AP ($\overrightarrow{AP}$) are created by connecting the pair of points in the opposite side of the rectangle APQR. Finally the square grids are achieved in term of points. The equations related to this paragraph are illustrated below:

The equation used to calculate the length of vector where the start point is $X_1(x_1, y_1, z_1)$ and the end point is $X_2(x_2, y_2, z_2)$, is shown in (e.q 3.7)

$$\left|\overrightarrow{X_1X_2}\right| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \qquad \text{(e.q 3.7)}$$

The equation uses to calculate the $X(x, y, z)$ is

$$\frac{\left|\overrightarrow{AX}\right|}{\left|\overrightarrow{AB}\right|} = \frac{m}{n} \qquad \Rightarrow \left|\overrightarrow{AX}\right| = \frac{m}{n}\left|\overrightarrow{AB}\right| \qquad \text{(e.q 3.8)}$$

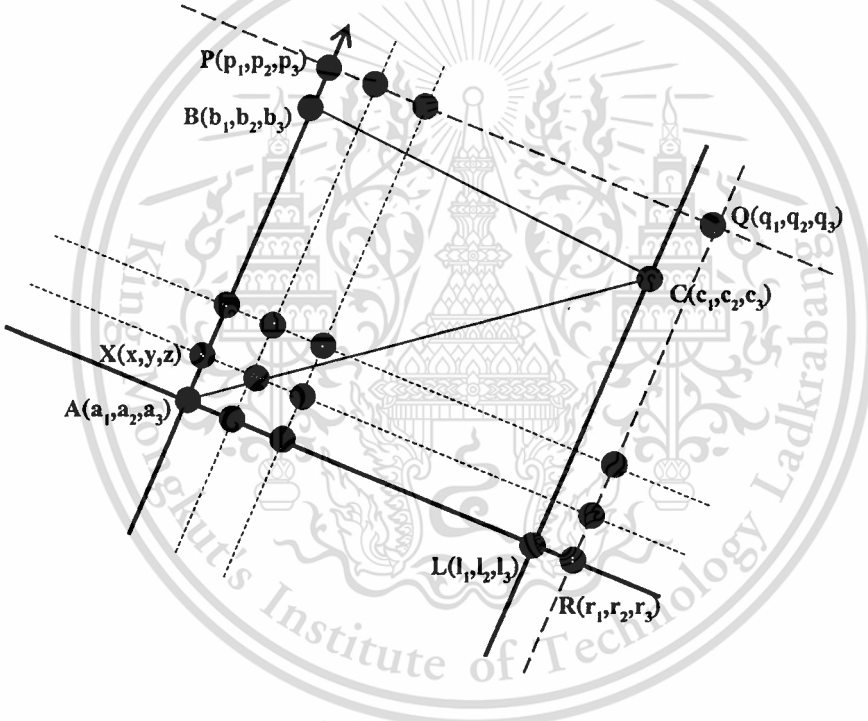


Figure 3.22: Calculation the value of the element of points' grid.

3.4.2: Definition of the geometry grids

The processes of defining the geometry grids are separated into four steps. First, calculate the center point in each cell grid (using (e.q 3.9)). The step is similar to the process that defines the grid points. As mentioned in section 2.5.1, according to the Fig. 3.23.b the geometry of square grid can be defined into three groups. First, if the center point is exactly inside triangle. This cell is defined as a *Cell_In* position (● sign). Second, if any center point locates outside the triangle and the line of the triangle intersect with its cell grid in both X, Y dimension. This cell is defined

as *Cell_Part* position (○ sign). Another center point is not one of two cases mentioned above, so this cell is defined as a *Cell_Out* position (⊗ sign). The method and the equations used to define all types of cells are assigned below:

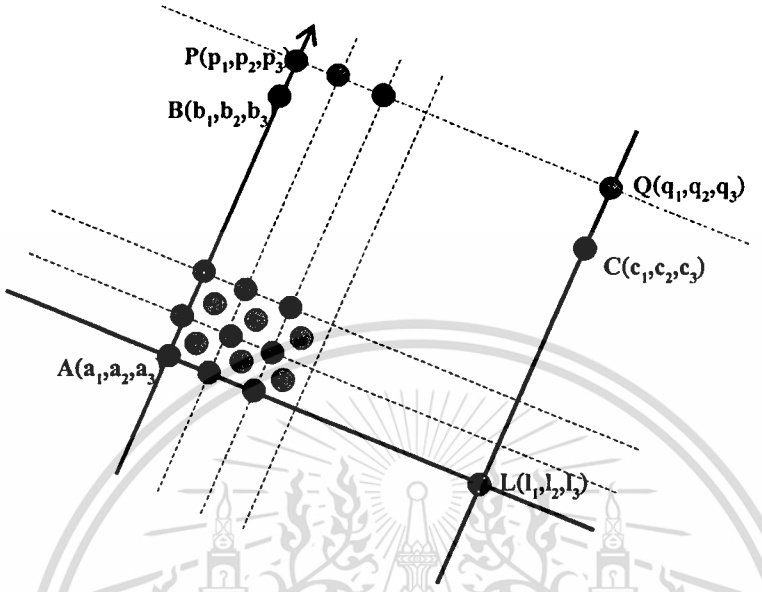


Figure 3.23.a: Calculation the value of the center points.

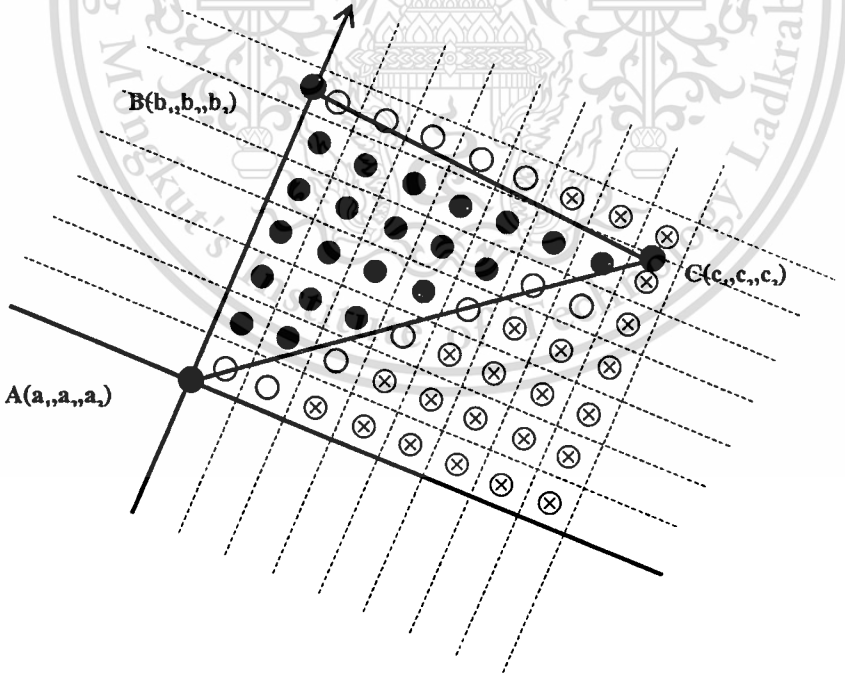


Figure 3.23.b: The definition of three types of the geometry grid.

The equation used to calculate the center point is: let $Y_{ij}(d\_row,d\_col)$ be the center point of cell in i^{th} row and j^{th} column where the coordinate of point Y_{00} is $(\frac{l\_cell}{2},\frac{l\_cell}{2})$ and

l_cell be the length of cell, the equation used to calculate the coordinate of the center point is illustrated in (e.q 3.9 & e.q 3.10).

$$d\_row_{i+1} = d\_ro_i + l\_cell \qquad \text{(e.q 3.9)}$$

$$d\_col_{j+1} = d\_col_j + l\_cell \qquad \text{(e.q 3.10)}$$

When $\vec{a}$ is (x_1, y_1, z_1) and $\vec{b}$ is (x_2, y_2, z_2), the equation used to calculate $(\vec{a} \times \vec{b})$ is:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix} = \begin{vmatrix} y_1 & z_1 \\ y_2 & z_2 \end{vmatrix} \vec{i} + \begin{vmatrix} x_1 & z_1 \\ x_2 & z_2 \end{vmatrix} \vec{j} + \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} \vec{k} \qquad \text{(e.q 3.11)}$$

When $\vec{a}$ is (x_1, y_1, z_1) and $\vec{b}$ is (x_2, y_2, z_2), the equation used to calculate $(\vec{a}.\vec{b})$ is:

$$\vec{a}.\vec{b} = (x_1 * x_2) + (y_1 * y_2) + (z_1 * z_2) \qquad \text{(e.q 3.12)}$$

The equation use to determine *Cell_In* position is: let's consider the point P in Figure 3.25.b, the point P locates inside the rectangle MNGK if it corresponds to this equation:

$$(\overrightarrow{AB} \times \overrightarrow{AP}).(\overrightarrow{BC} \times \overrightarrow{BP}) \geq 0 \quad and \quad (\overrightarrow{AB} \times \overrightarrow{AP}).(\overrightarrow{BC} \times \overrightarrow{BP}) \geq 0 \qquad \text{(e.q 3.13)}$$

If the point P does not correspond to e.q 3.13, it means that the point P outside ΔABC. To check *Cell_Part* position: let's consider the size of the cell (MN and NK) and line BC of ΔABC, if $(MN) \cap (BC)$ *and* $(KN) \cap (BC)$, so the rectangular MNGK is the *Cell_Part* position. The equation used checks the intersection between two lines is e.q 3.14.

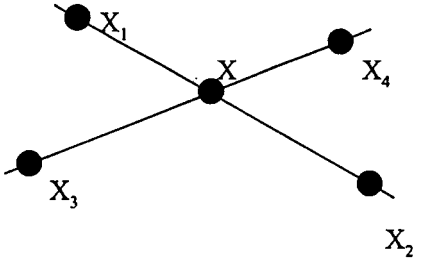


Figure 3.24.a: Lines intersection

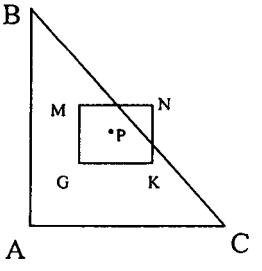


Figure 3.24.b: Checking point

$$\overrightarrow{X_1X_3}.(\overrightarrow{X_1X_2} \times \overrightarrow{X_3X_4}) = 0 \qquad \text{(e.q 3.14)}$$

3.4.3: Cell communication in CA

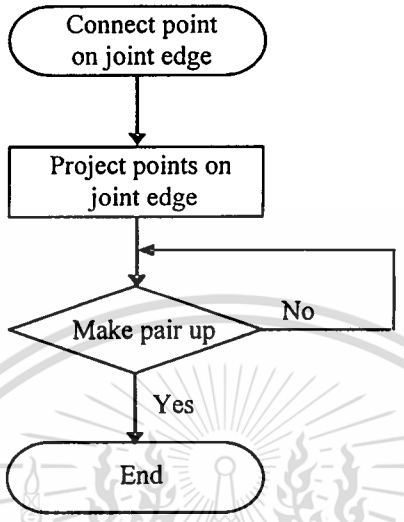


Figure 3.25: Flow chart for connecting point on joint edge

A cell in a triangle can communicate with their surrounding eight neighbor cells only. However some cells have less than eight neighbors, especially the cells at the edge of a triangle. These cells have to connect to the neighbor cells, which are in the opposite triangle. It means that, the cells on the joint edge of two opposite triangles are sending and receiving information between each other. The cells on the opposite side of the triangles are paired up, and the transmission of each pair is determined. The process of the data transmission of the cells can be described in terms of mathematics with corresponding Figure 3.27. The cell on the edge of the triangle will communicate to the cell in the nearest triangle, if this cell was an intersection with its triangle line and this line is the joint edge of two triangles (for example, $\overrightarrow{A_2B_2}$ and $\overrightarrow{C_1A_1}$ are the joint line of $\Delta A_1B_1C_1$ and $\Delta A_2B_2C_2$ in Figure 3.26). After that the cell's center points in both two triangles are projected on the joint edge (using e.q 3.15) and calculated the shortest distance of two cell's center points for determining a pair of them (using e.q 3.16). Figure 3.26 shows the determination of cell's center point between two triangles. After determination of the pair of cell's center point between two triangles, the data is transmitted over the 3D model. The general cell structure is illustrated in Figure 3.28 and the equations related to this paragraph are illustrated below:

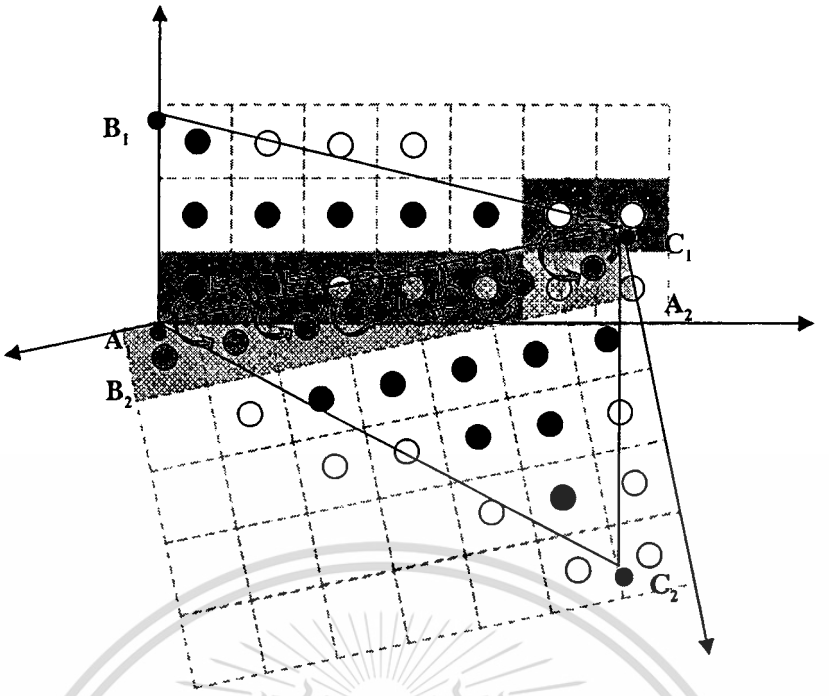


Figure 3.26: Data transmission on joint edge

In Figure 3.26, an error occurs at both end points, some cells will receive information twice and others will only send data. Actually, an error is often generated during data transfer on edges and happens most frequently at extremities (triangle vertices). However, the experiment and the practical observations prove that this effect can be negligible if the size of grid is small enough. Moreover, if the triangles are well organized (making the projected grid continuous), the data transmission does not become redundant and therefore the problem disappears.

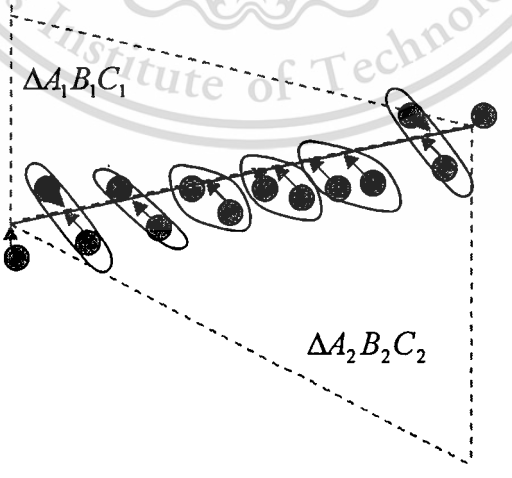


Figure 3.27: Projection of the center point on joint edge

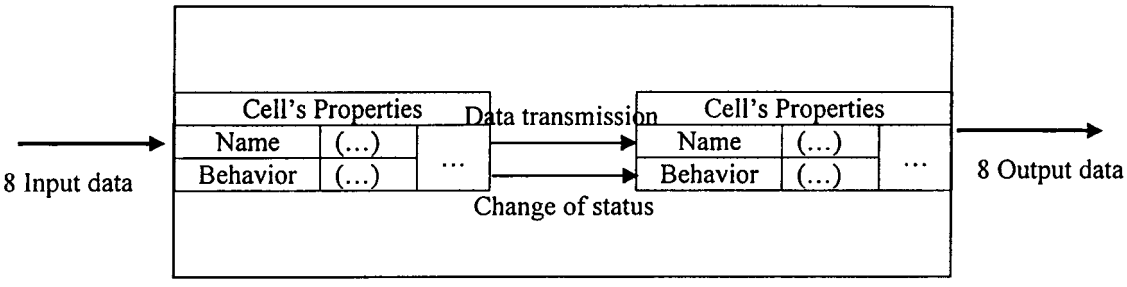


Figure 3.28: The general structure of a cell

In figure 3.29, the point $M_1(x1, y1, z1)$ is projected on the line B_2C_1, and the point M_1' (x'_1,y'_1,z'_1) is the new position of the point $M_1(x1, y1, z1)$ on the Line B_2C_1, the coordinate of point M_1' can be calculated by the equation in (e.q 3.15):

$$\overrightarrow{C_1B_2} \perp \overrightarrow{M_1'M_1} \quad \Rightarrow \quad \overrightarrow{C_1B_2} \cdot \overrightarrow{M_1'M_1} = 0 \qquad \text{(e.q 3.15)}$$

The distance from point M_1' (x'_1,y'_1,z'_1) to the point $N_1'(a'_1,b'_1,c'_1)$ (in Fig. 3.28)can be calculated by the equation in (e.q 3.16):

$$d(M_1', N_1') = \sqrt{(x_1' - a_1')^2 + (y_1' - b_1')^2 + (z_1' - c_1')^2} \qquad \text{(e.q 3.16)}$$

The end point N_3 (n_1,n_2,n_3) is projected on the line B_2C_1 by defining its vector. The vector N_3B_2 ($\overrightarrow{N_3B_2}$) is created where the N_3 is a start point and B_2 (b_1,b_2,b_3) is end point, and B_2 is a new position of the N_3 on the line B_2C_1. The equation used to project the N_3 point is:

$$\overrightarrow{N_3B_2} = (b_1 - n_1, b_2 - n_2, b_3 - n_3) \qquad \text{(e.q 3.17)}$$

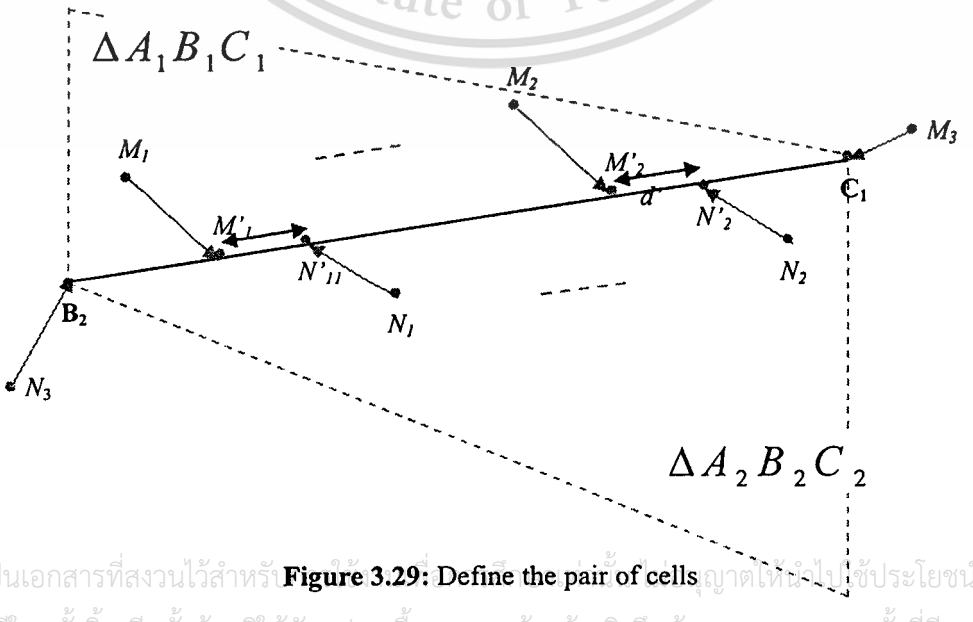


Figure 3.29: Define the pair of cells

CHAPTER 4

SIMULATION AND RESULT

4.1 3D Models for simulation

The 3D models which are used for the simulation are the .ase extension file. They are exported from 3Ds max program in section 3.3. As mentioned in section 3.3, the 3D models in the first group are the 3D models that have been directly created by 3Ds max and are processed to reduce the redundancy of faces. The models' faces are well arranged making the structures simple and easy to render. In contrast, the structures of the 3D models in the second group cannot be altered. As a result the number of triangle faces cannot be reduced. The 3D models in this group are very complicated. They are made from a large number of very small triangle faces. This causes computing and rendering times to be very high. The figure 4.1 and figure 4.2 show the wire frame rendering in 3Ds max program where the two examples models are in the first group and the last group respectively.

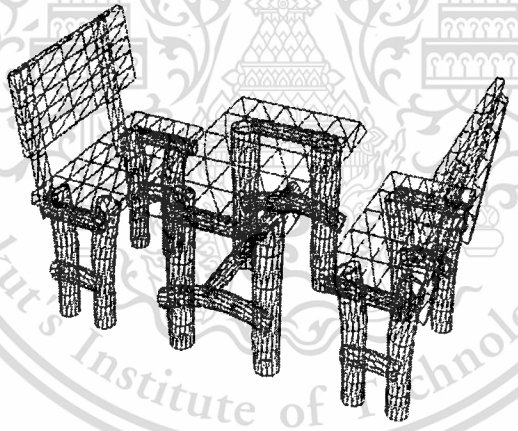


Figure 4. 1: The table model in wireframe rendering

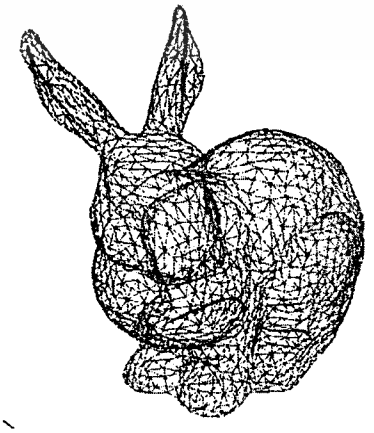


Figure 4. 2: The Stanford Bunny model in wireframe rendering

4.2 Equipments for simulation:

This research has simulated the weathering effect on wood by using C programming with the OpenGL libraries and Windows API. The equipments used for the simulation are listed below:

1. CPU: Pentium(R) D 2.66 GHz
2. RAM: 512 MB
3. Hard disk: 80 GB
4. VGA memory: 256 MB
5. Operating System (OS): Windows XP Professional
6. Program is used to develop: Ms Visual Studio C++ 2005

4.3 Simulation:

The steps to simulate the weathering effect on wood are separated into three main parts, each part is a distinguished responsibility. The first part is conversion of the 3D model to 3DSCA models. This is done by creating grid, computing the center point of grid, an intermediate point for each face. The second part is for the computation of the connection of points which are in between opposite faces. The final part involves the application of the CA rules, mapping the wood texture on the wooden objects surface and the demonstration of the animation result. The programming1, programming2, and programmig3 are the programming for each part where the algorithm for each programming, and the explanation and the result of each part are illustrated below:

Algorithm of Programming1: (The source code in Appendix A)

1. Load A, B, C vertices for each face
2. Compute grid, center point of grid, intermediate point for each face.
3. Save the computing result as .txt extension files

Algorithm of Programming2: (The source code in Appendix B)

1. Load the TXT files in Programming1
2. Project the center point of cell at the joint edge of two opposite triangle to the joint edge.

3. Compute the shortest distance of the point which is different triangle, and make paired up or make a link between two points.
4. Save the link result as .txt extension files0

Algorithm of Programming3: (The source code in Appendix C)

Subprogramming3_1:

1. Define $f(x, y)$ function to generate texture
2. Define GL_OBJECT_PLANE or GL_SPHERE_MAP function to map the texture on the wooden objects
3. Define GL_MODULATE function to modulate the color.
4. Define reference plane to generate coordinate of the texture

Subprogramming3_2:

1. Load the TXT files in Programming2
2. Compute the neighborhood for each cell
3. Define the range of cell's values for specifying the color
4. Define the CA rules
5. Define the GL_TRIANGLE_FAN and GL_TRIANGLE_STRIP arguments for rendering.

4.3.1 Constructing the 3DSCA model

This section is the first part of the simulation, it applies the technique and mathematical functions illustrated in section 3.4.1, 3.4.2 for creating the 3DCSA model. The programming1 is used to import the vertices of the 3D model in the ASE file. First, the programming1 starts to load the first face, computes the cell grid, center point of grid, and intermediate for each face, then defines the properties for each cell (*Cell_in*, *Cell_Part* or *Cell_out*). After the computation of the first face is finished, the next face is continuously loaded. The programming1 exits when the last face of the 3D model has been computed. The computing results are stored in TXT files and will be used in the next part where the graphics results are drawn by defining the GL_POINTS, GL_TRIANGLES, GL_LINE_LOOP arguments in OpenGL. The computing processes in this part are very time consuming. The length of time depends on the amount of the model's faces and

the size the of model's faces. The graphics results for each 3D model are listed in the following figures:

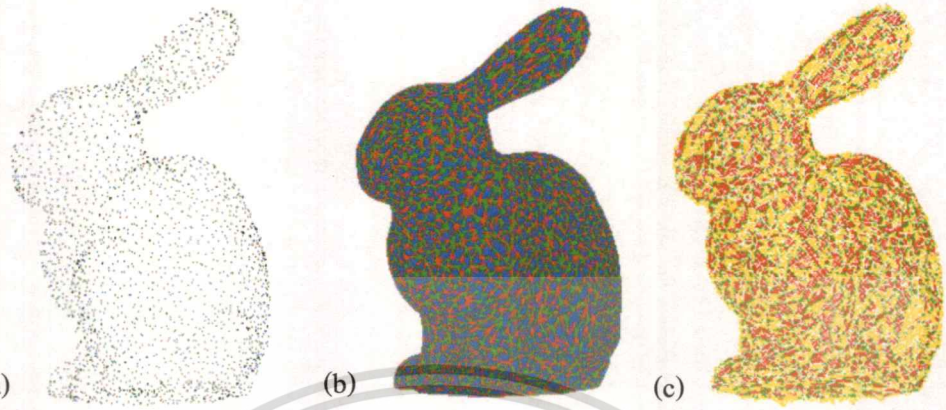


Figure 4. 3: Drawing the Bunny model

(a). Drawing the Bunny model by argument: GL_POINTS (2,503 vertices)

(b). Drawing the Bunny model by argument: GL_TRIANGLES (4,968 faces)

(c). Drawing the grid point, center point and intermediate point of grid of the Bunny model (grid size: 0.03 and 389,942 cells, time for drawing less than 2 minutes)

Figure 4. 4: Drawing the table model

(a). Drawing the table model by argument: GL_POINTS (2,372 vertices)

(b). Drawing the table model by argument: GL_TRIANGLES (4,608 faces)

(c). Drawing the grid point, center point, and intermediate point of grid of the table model (grid size: 0.3 and 275,398 cells, time for drawing less than 2 minutes)

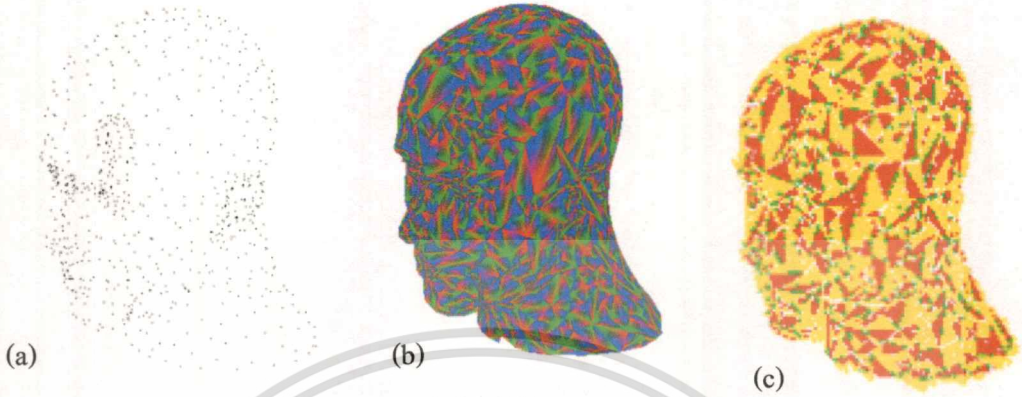


Figure 4. 5: Drawing the head model

(a). Drawing the head model by argument: GL_POINTS (689 vertices)

(b). Drawing the head model by argument: GL_TRIANGLES (1,355 faces)

(c). Drawing the grid point, center point, and intermediate point of grid of the head model (grid size: 0.04 and 18,978 cells, time for drawing less than 2 minutes)

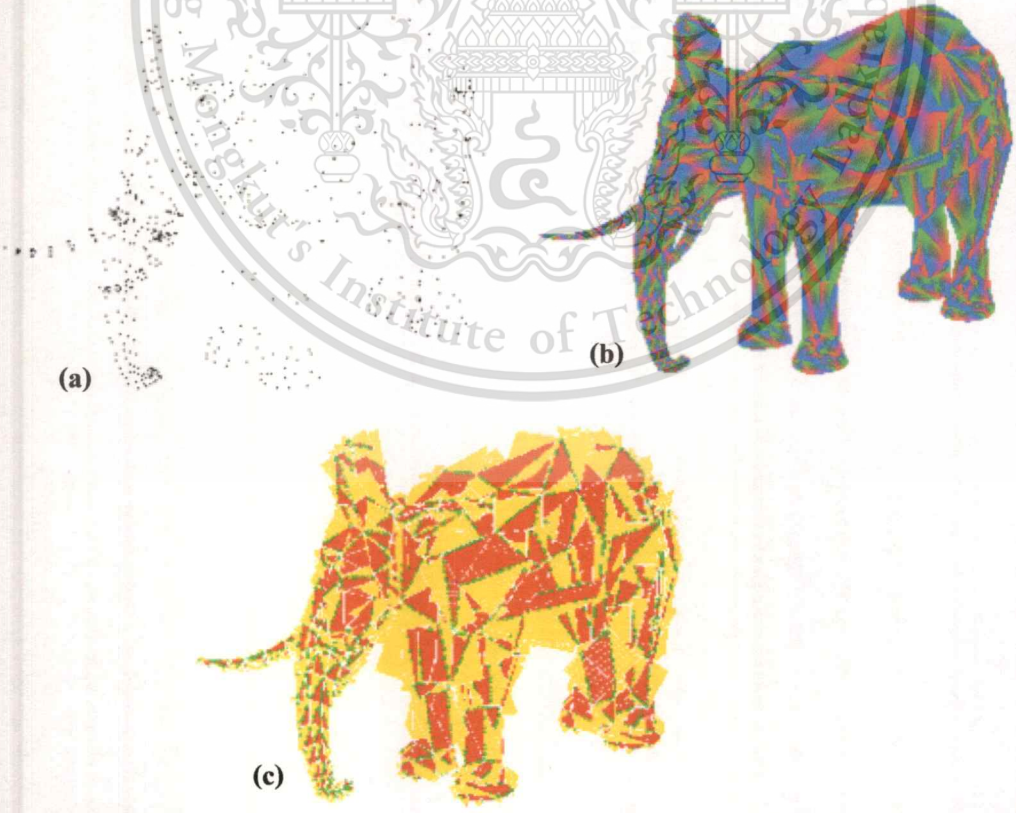


Figure 4. 6: Drawing the elephant model

(a). Drawing the elephant model by argument: GL_POINTS (623 vertices)

(b). Drawing the elephant model by argument: GL_TRIANGLES (1,148 faces)

(c). Drawing the grid point, center point, and intermediate point of grid of the elephant model (there are 57,1398.00 cells, time for drawing less than 25 minutes)

4.3.2 Connecting points on joint edge

From the previous section, the 3DSCA models are constructed. This means that all faces of the 3D models are converted to a group of cells. The computing results from the first part are used in the second part. The mathematical functions in section 3.4.3 are applied in this section. They connect the cell at the joint edge to another cell in opposite face. The computing results are also saved as TXT files and are used in the final part of the simulation. The graphics results for each model can be captured in figure 4.7.

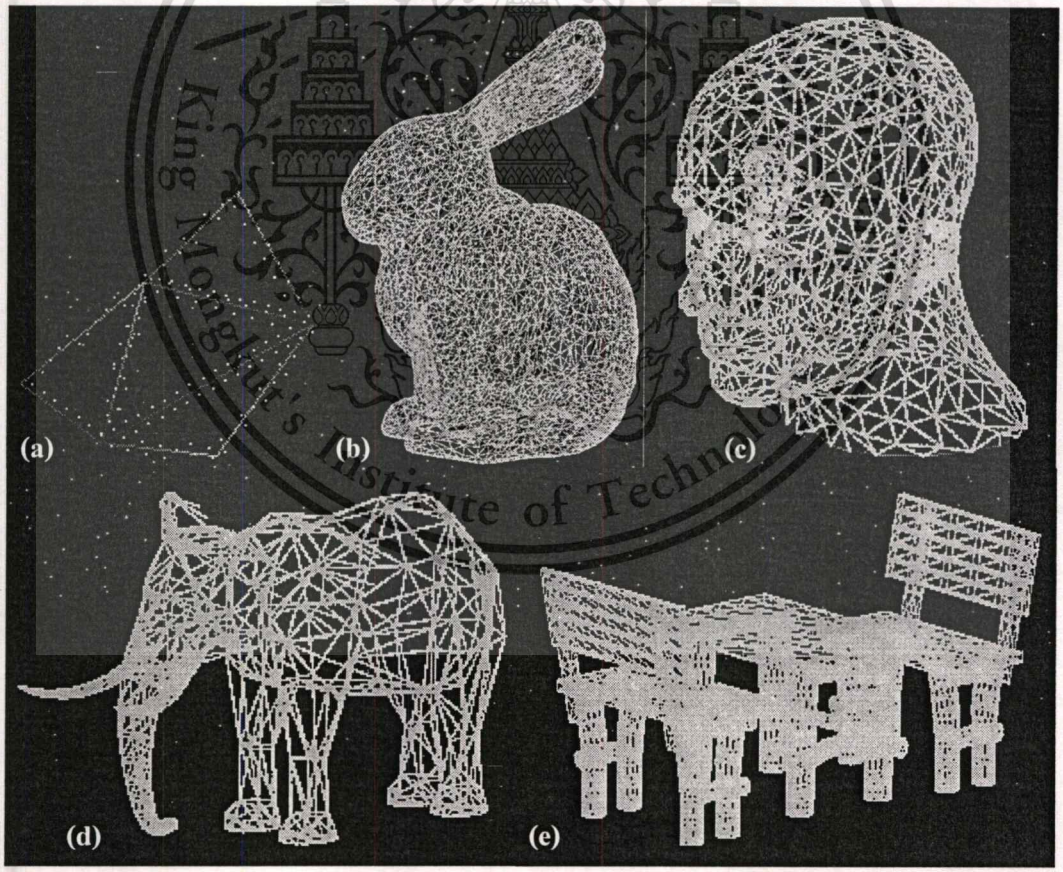


Figure 4. 7: Drawing the link point of the 3D models

(a) Drawing the link point in between four faces of the 3D model

(b) Drawing the link point of the bunny model (time for computing and drawing: less than 0.5 minutes)

(c) Drawing the link point of the head model (time for computing and drawing: less than 0.5 minutes)

(d) Drawing the link point of the elephant model (time for computing and drawing: less than 0.5 minutes)

(e) Drawing the link point of the table model (time for computing and drawing: less than 0.5 minutes)

4.3.3 Applying CA rules

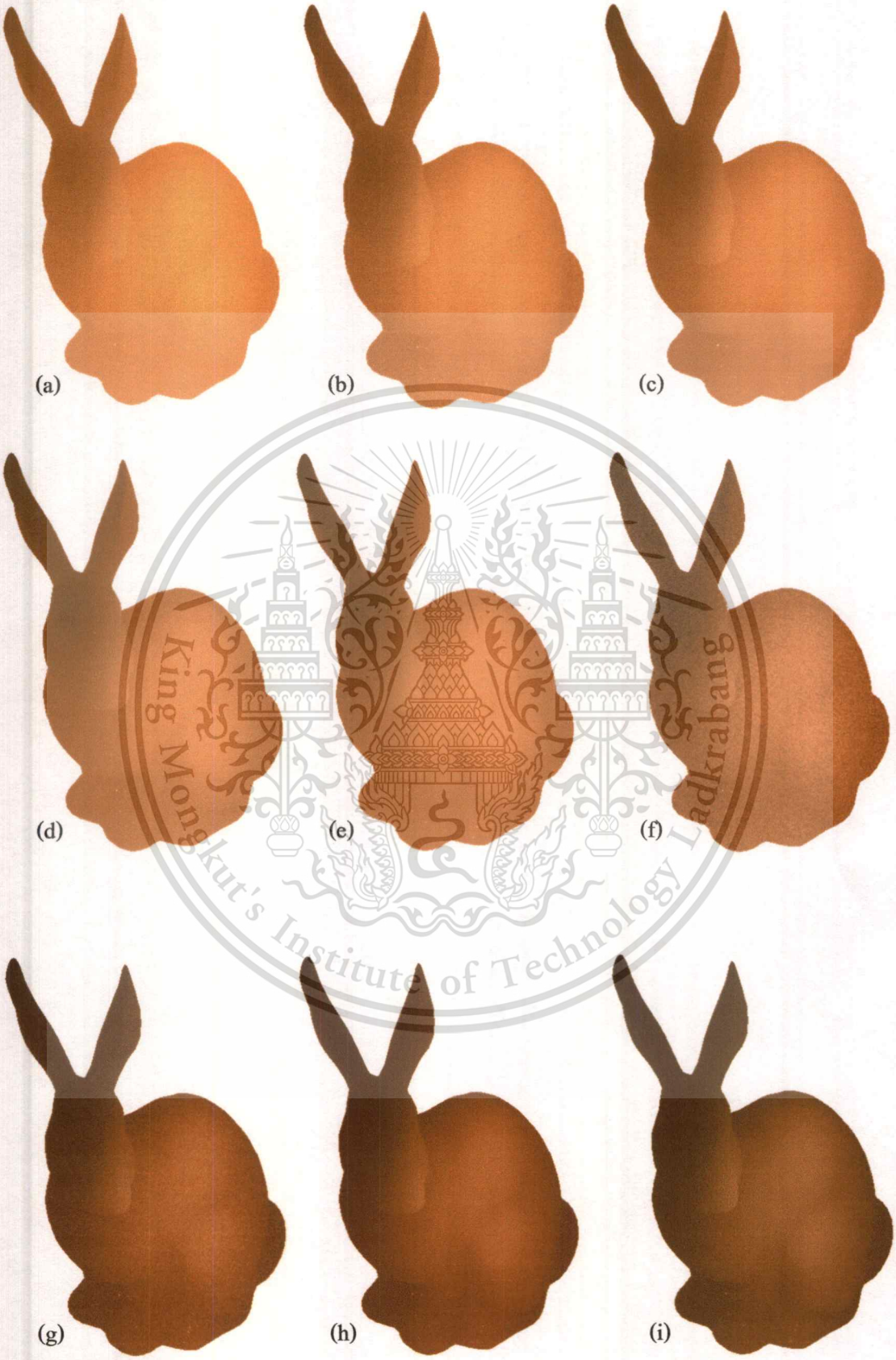
This section applies the two rules of CA that were outlined in section 3.2 onto the 3DSCA models. The computing results in section 4.3.2 are also used to determine the neighborhoods of cells. As section 3.2 illustrated that each cell has two possible statuses, and each status is defined by the cell's value. In this simulation, the values of cells are used to define the color of cells. They are set in many ranges and each range is defined in different color. The first step to render CA rules, the alive cells are random, which means that the cell's values are random. With each time step the cells' values are increased by one. The cells' color is changed to another color if the cells' values reach to another range. The smoothness of the change of the color depends on the color's values or cell's values in each range. To show the graphics results, two arguments GL_TRIANGLE_FAN and GL_TRIANGLE_STRIP in OpenGL are used to render the expanding of CA rules. The algorithm for rendering these arguments is shown below:

```
row_min // minimize position of row
row_max // maximize position of row
j // position of column
if(row_min[j]>row_min[j+1])
{
        for(p_pos = row_min[j+1];p_pos < row_min[j];p_pos++)
                Triangle Fan
        if(row_max[j]>row_max[j+1])
                for(p_pos = row_min[j];p_pos <= row_max[j+1];p_pos++)
                        Triangle Strip
```

```
                for(p_pos = row_max[j+1];p_os <= row_max[j];p_pos++)
                        Triangle Fan
        Else
                for(p_pos = row_min[j];p_pos <= row_max[j];p_pos++)
                        Triangle Strip
                if(row_max[j]<row_max[j+1])
                        for(p_pos = row_max[j]+1;p_pos <= row_max[j+1];p_pos++)
                                Triangle Strip
                else if(row_max[j]== row_max[j+1])
                        Triangle Strip
}
else if(row_min[j] == row_min[j+1])
{
        if(row_max[j]==row_max[j+1])
                for(p_pos = row_min[j];p_pos <= row_max[j];p_pos++)
                        Triangle Strip
        else if(row_max[j]<row_max[j+1])
                for(p_pos = row_min[j];p_pos <= row_max[j];p_pos++)
                        Triangle Strip
        else if(row_max[j]>row_max[j+1])
                for(p_pos = row_min[j];p_pos <= row_max[j+1];p_pos++)
                        Triangle Strip
                for(p_pos = row_max[j+1]+1;p_pos <= row_max[j];p_pos++)
                        Triangle Fan
}
else
        Triangle Strip
```

The graphics results of this section are animatedly presented, and they can be captured in following figures.

- **Rule 25/2356**



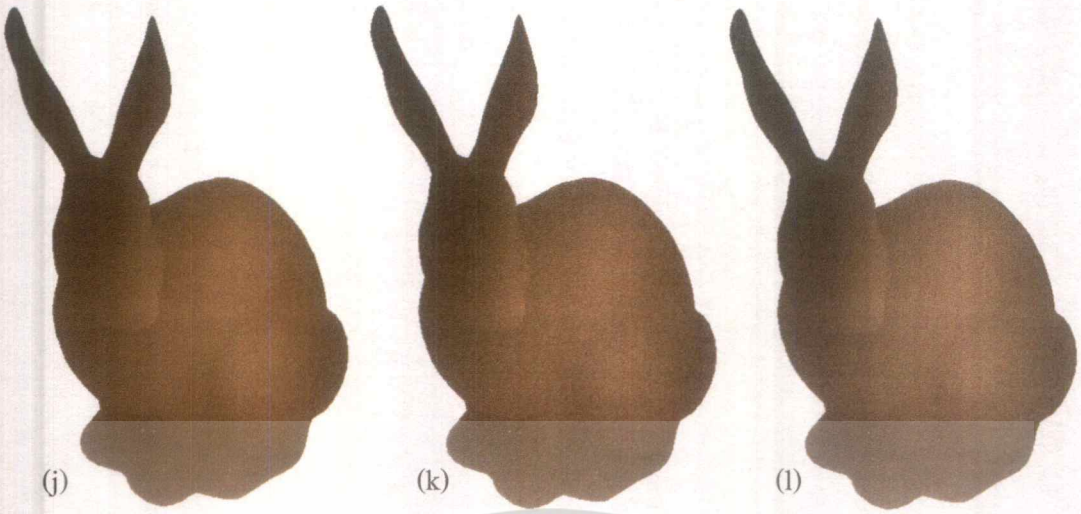
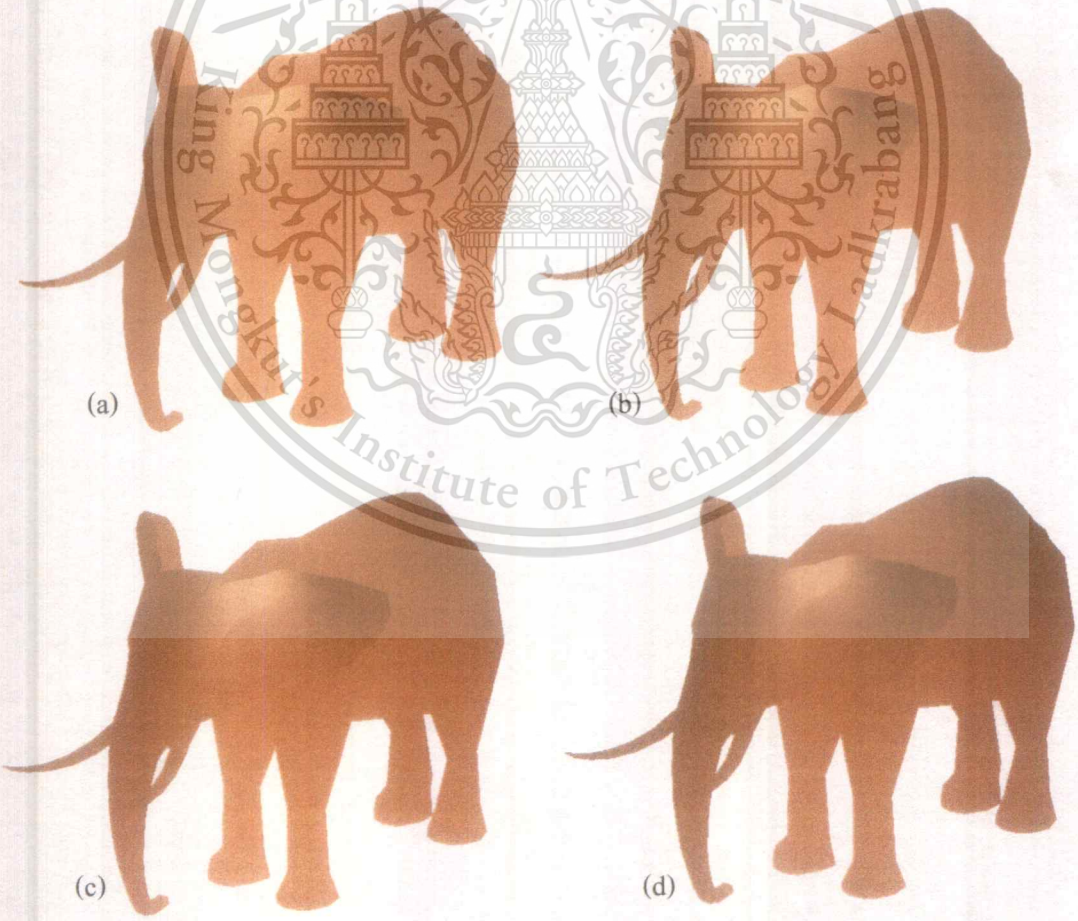


Figure 4. 8: The graphics result of applying rule 25/2356 on the Bunny model



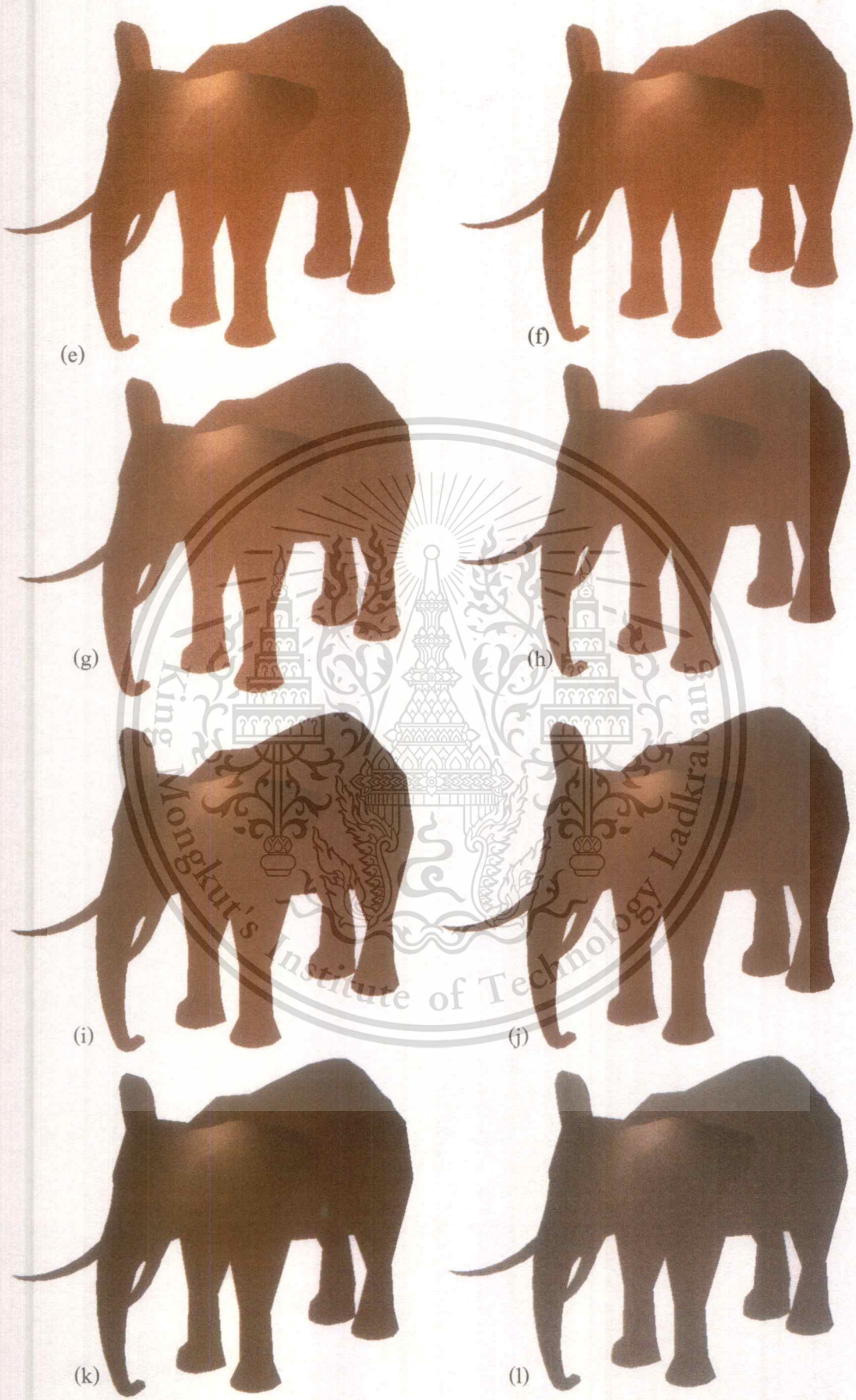
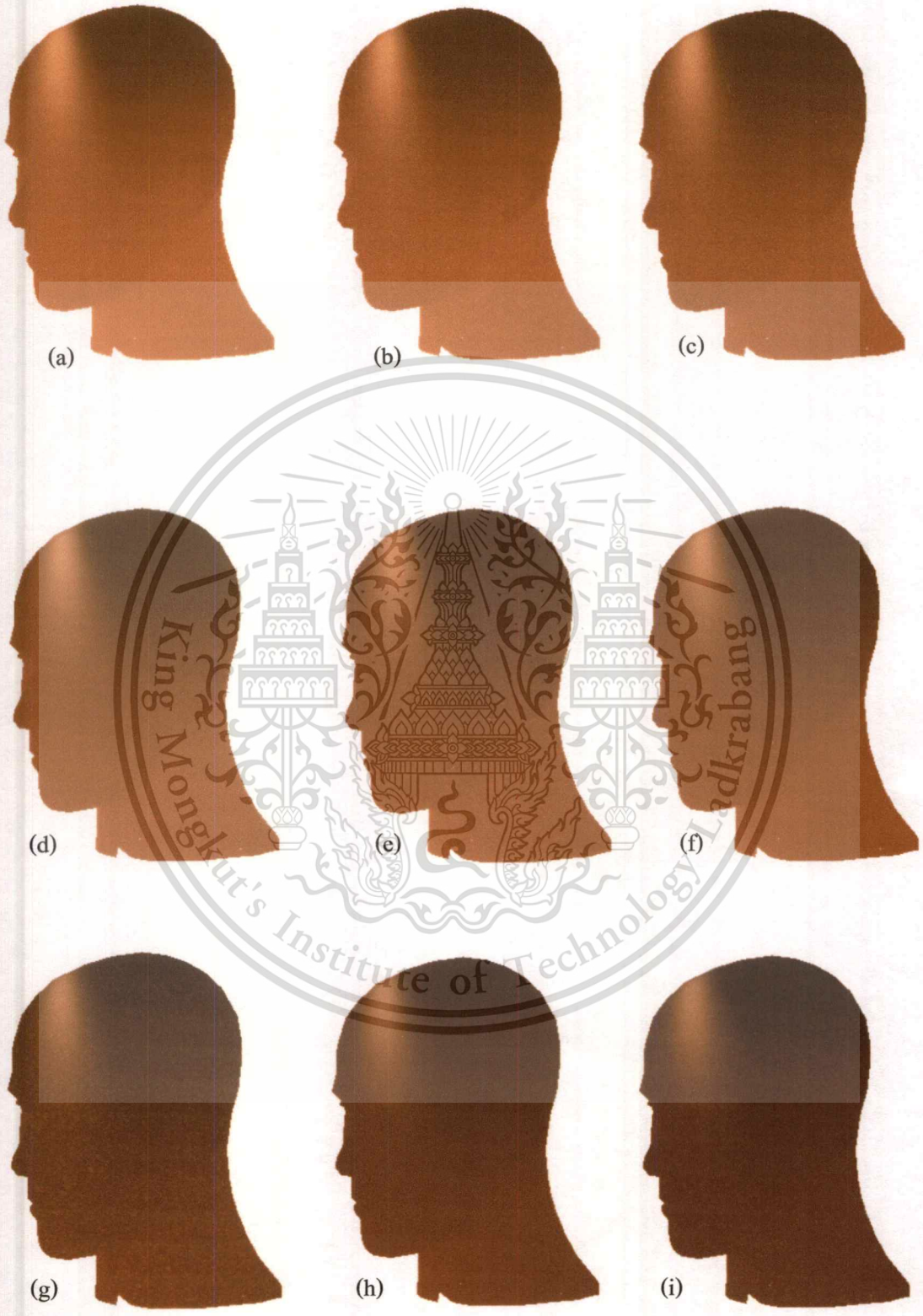


Figure 4. 9: The graphics result of applying rule 25/2356 on the elephant model

- **Rule 25/ 357**



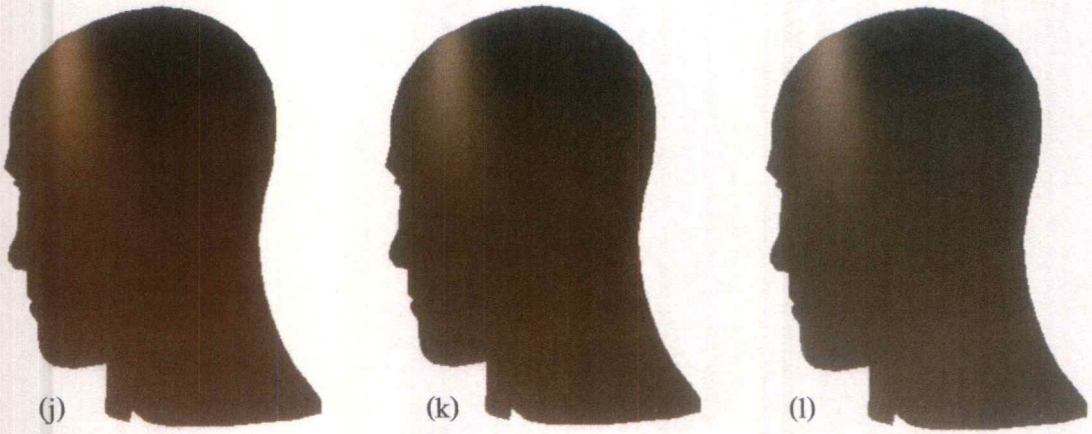
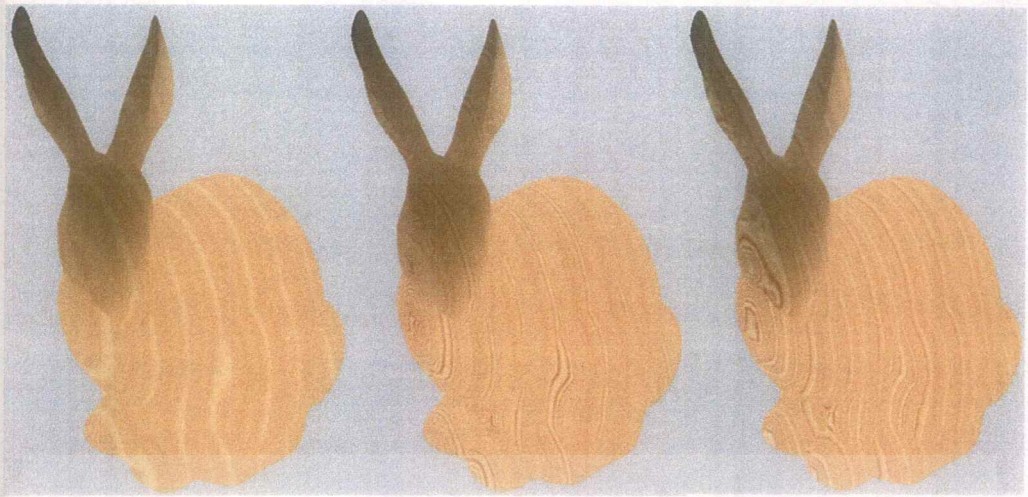


Figure 4. 10: The graphics result of applying rule 25/357 on the head model

Figure 4. 11: The graphics result of applying rule 25/357 on the table model

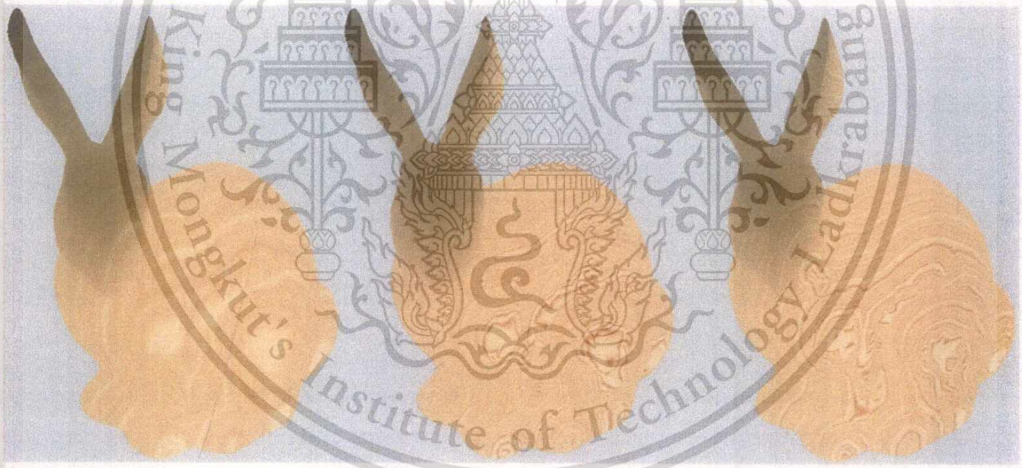
4.3.4 Wood texture mapping on wooden objects

As mentioned in the previous section, to render by applying two CA's rules show very good result mentioned in the previous section, but the surfaces of the wooden objects are not similar to the pattern of wood in real world. To improve the realistic in the previous section, this section generated wood textures as presented in section 3.1, and used GL_OBJECT_PLANE and GL_SPHERE_MAP functions in OpenGL to map these textures on the wooden objects. The texture coordinates are automatically generated by defining the reference planes as mentioned in section 2.5.3.5. The examples of texture mapping in each function on the 3D models are shown below:



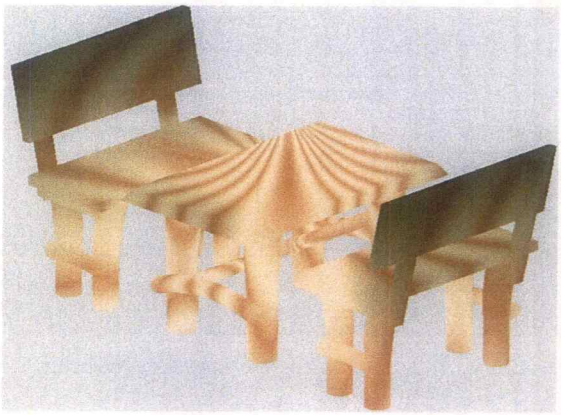
a. $f(x,y)=x^2-y^2$ b. $f(x,y)=x^2$ c. $f(x,y)=\cos e(3.14x)$

Figure 4. 12: Texture mapping by GL_OBJECT_PLANE function on the Bunny model with reference plane (p_1, p_2, p_3, p_4) = (1, 0, 0, 0), and the $f(x,y)$ is the function that is used to generate the pattern of wood texture.



a. $f(x,y)=x^2-y^2$ b. $f(x,y)=x^2$ c. $f(x,y)=\cos e(3.14x)$

Figure 4. 13: Texture mapping by GL_OBJECT_PLANE function on the Bunny model with reference plane (p_1, p_2, p_3, p_4) = (0, 1, 0, 0), and the $f(x,y)$ is the function that is used to generate the pattern of wood texture.



$$f(x,y)=x^2+y^2,\ y\geq 0$$

Figure 4. 14: Texture mapping by GL_SPHERE_MAP on the table model.
The $f(x,y)$ is the function that is used to generate the pattern of wood texture.

When mapping the texture on the surface object, the CA color of wooden objects and the pattern of texture are combined. As mentioned in section 2.5.3.3, there are four ways to combine the CA color with texture to get final pixel color, texture replaces (decal or replace) the color, texture blends the color, or texture modulates the color. This simulation used the GL_MODULATE function to map the wood texture on the wooden objects. From this function, the CA color is multiplied by a corresponding value in the texture to determine final pixel color. The result of texture modulates the CA color is shown in figure 4.15.

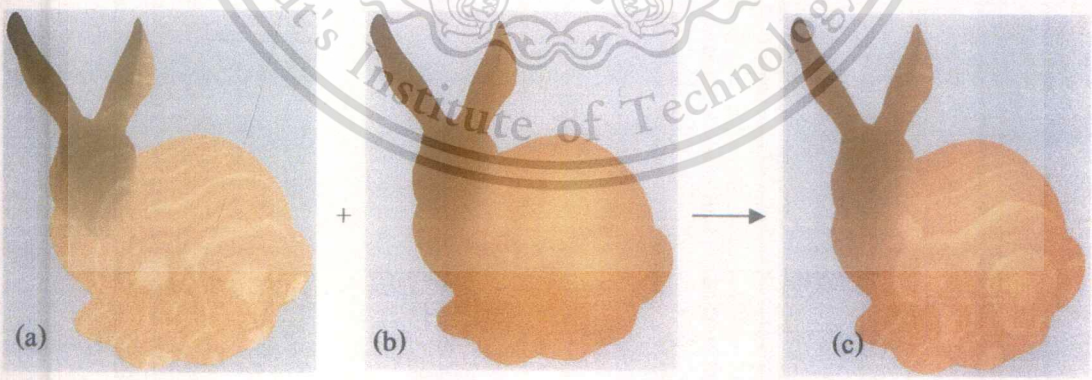


Figure 4. 15: Texture mapping and modulation result

(a) RGB texturing only

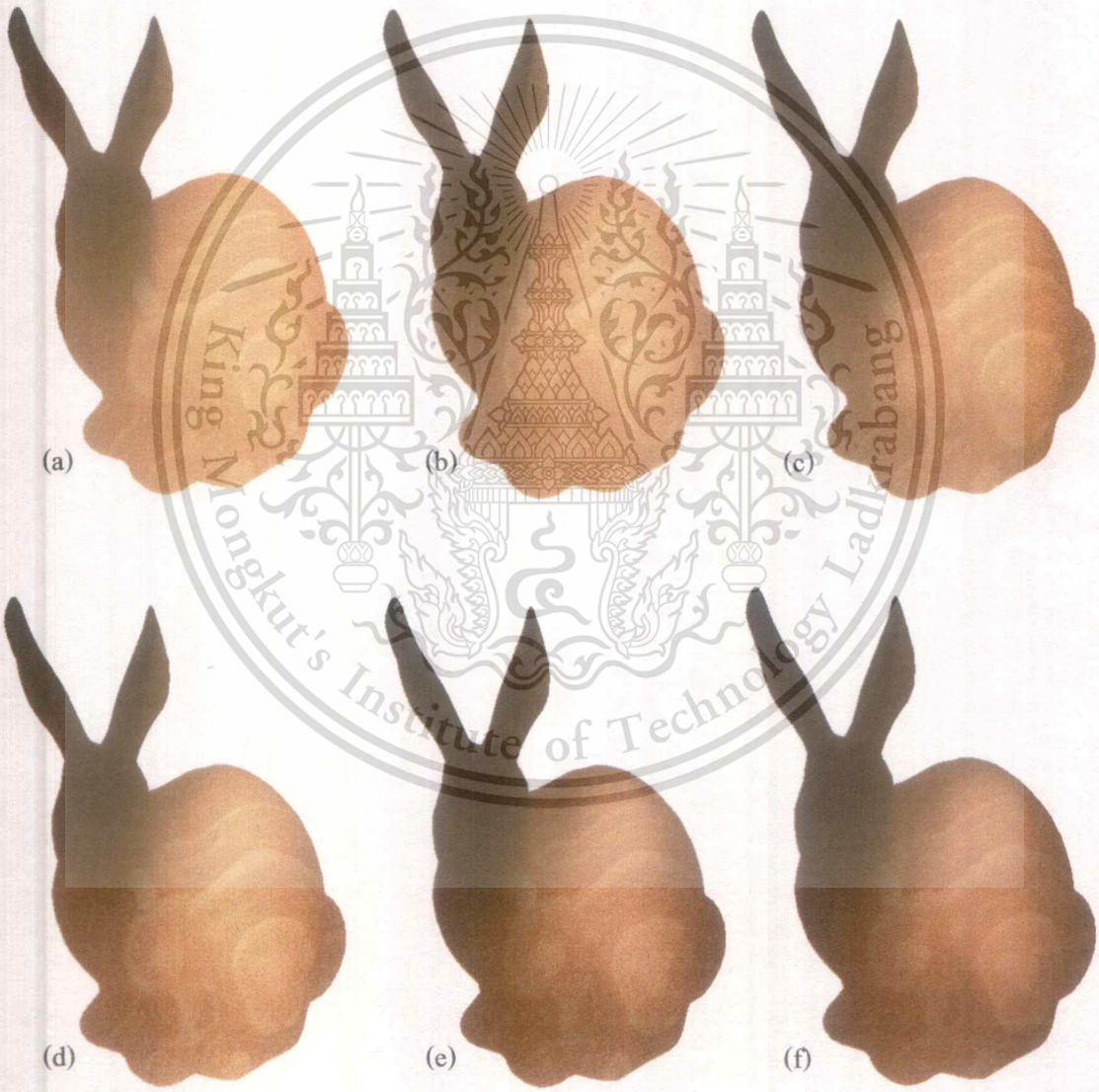
(b) CA color only

(c) Texture modulated by CA color

4.4 Result

After the CA's rules and the wood textures are simultaneously applied to the last part of the simulation, the results are more realistic as shown in figure 4.16, 4.17, 4.18, and 4.19 when compare with the results in section 4.3.3. It is clear that when applying both the 3DSCA method and the texture mapping. It is more realistic than applying the 3DSCA method alone. The results of the weathering simulation on wooden objects for each CA's rules are illustrated below:

- **Rule 25/ 2356**



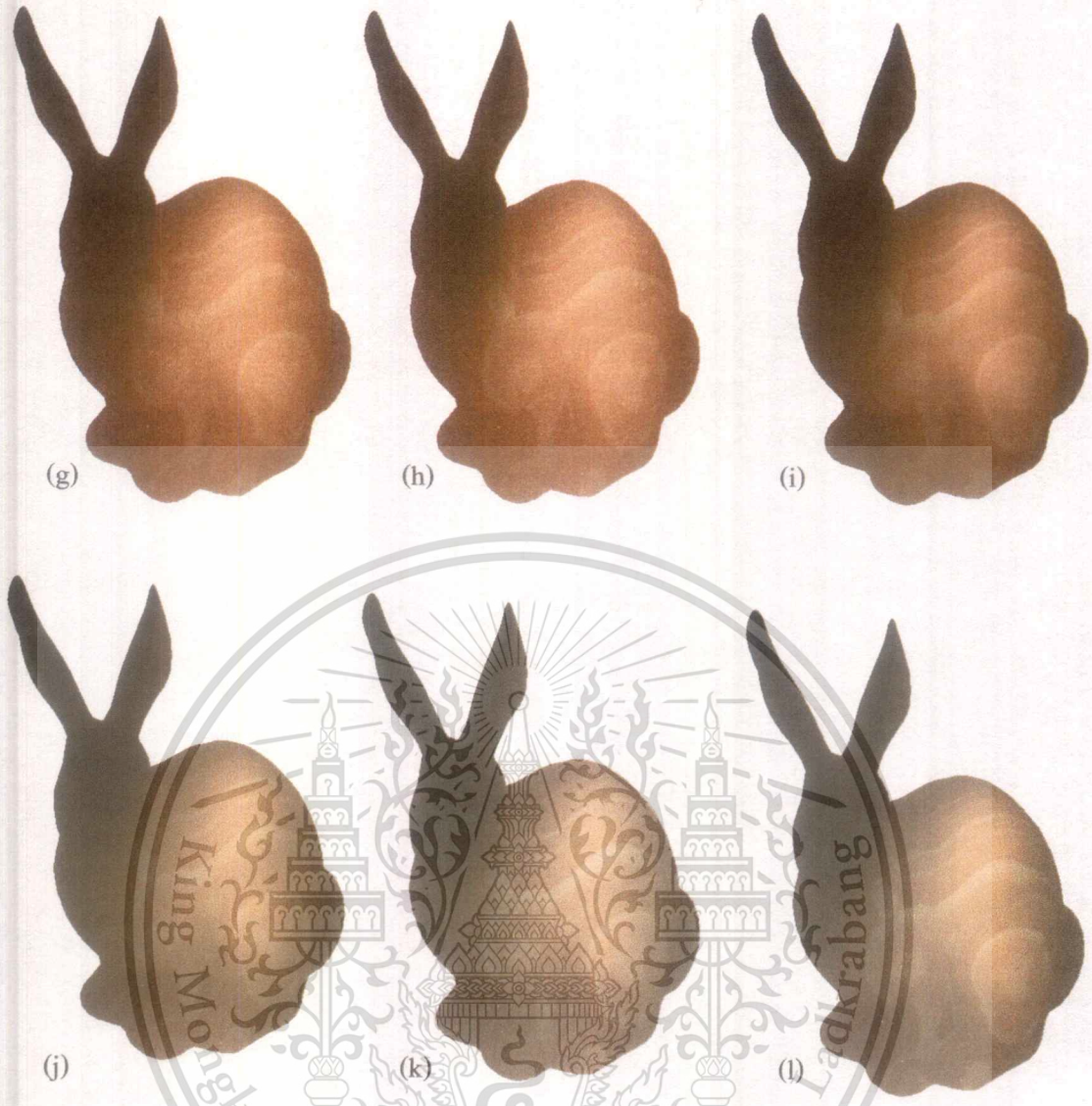


Figure 4. 16: The weathering simulation on the Stanford Bunny model by the rule 25/2356 (CA steps: 432, Time for complete rendering: less than 7 minutes)

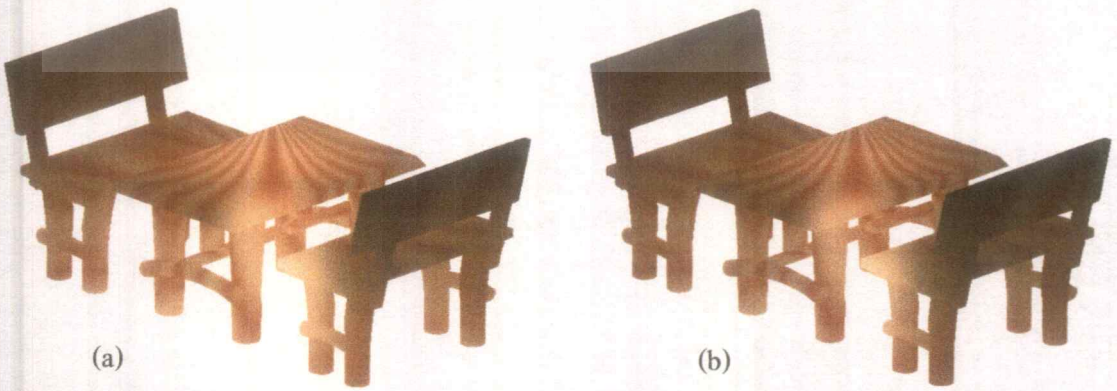
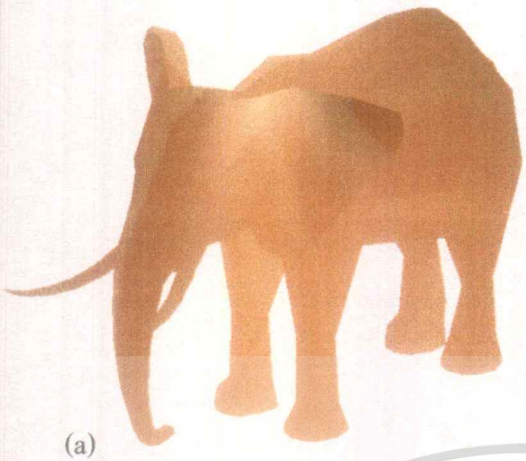


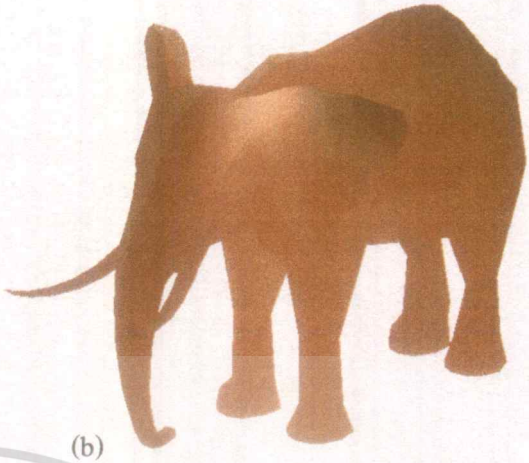
Figure 4. 17: The weathering simulation on the table model by the rule 25/2356

(CA steps: 432, Time for complete rendering: less than 6 minutes)

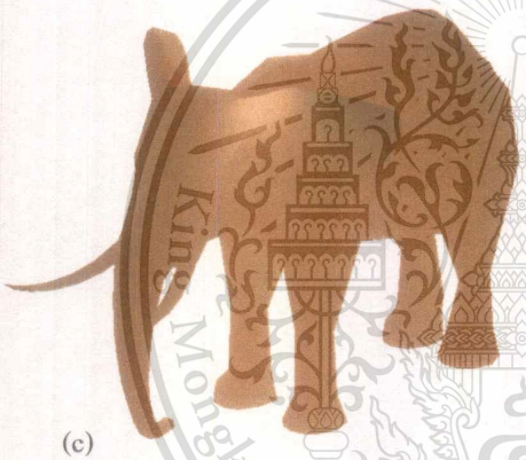
- **Rule 25/357**



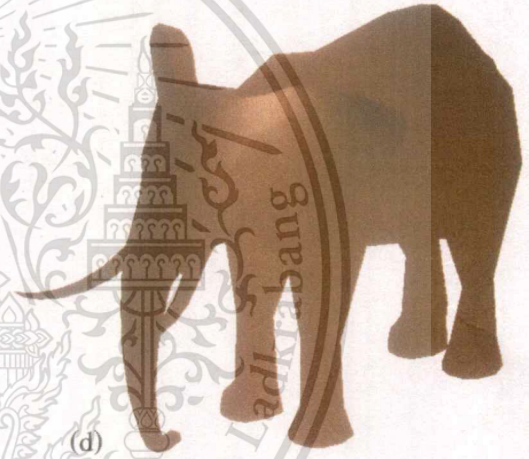
(a)



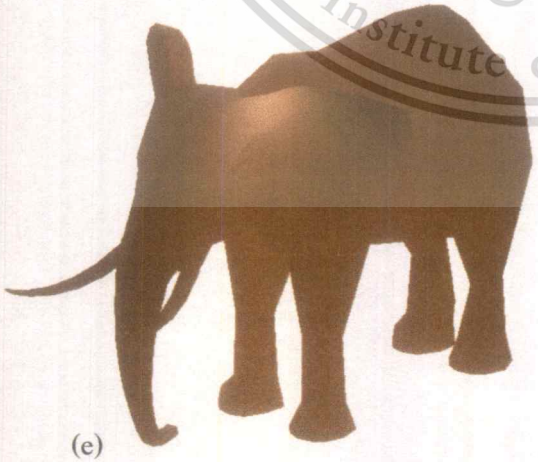
(b)



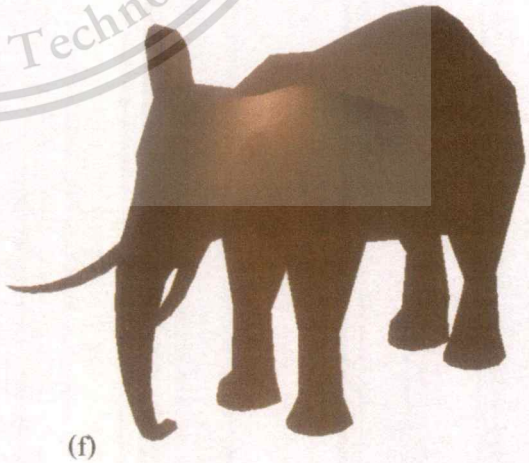
(c)



(d)



(e)



(f)

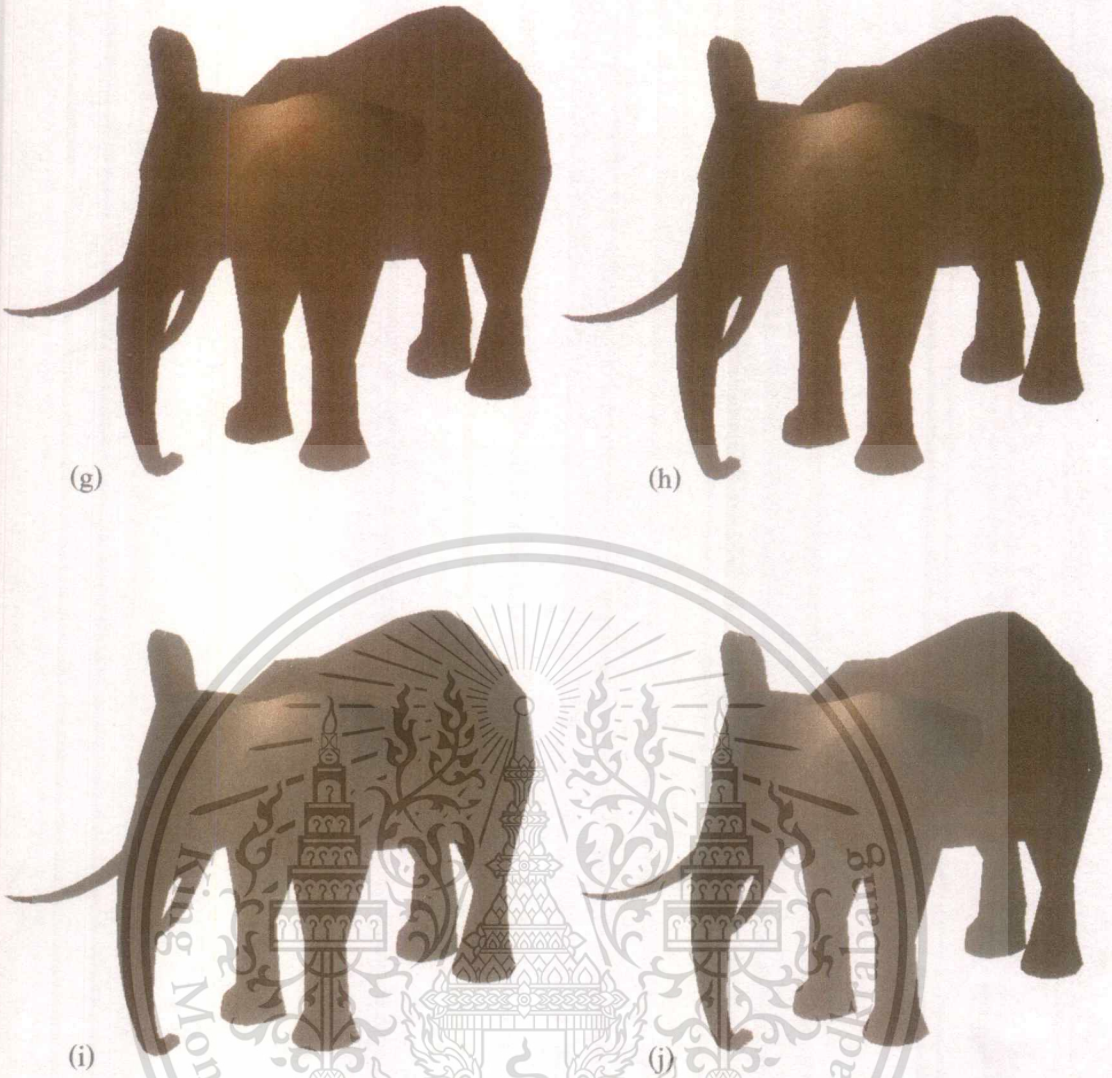
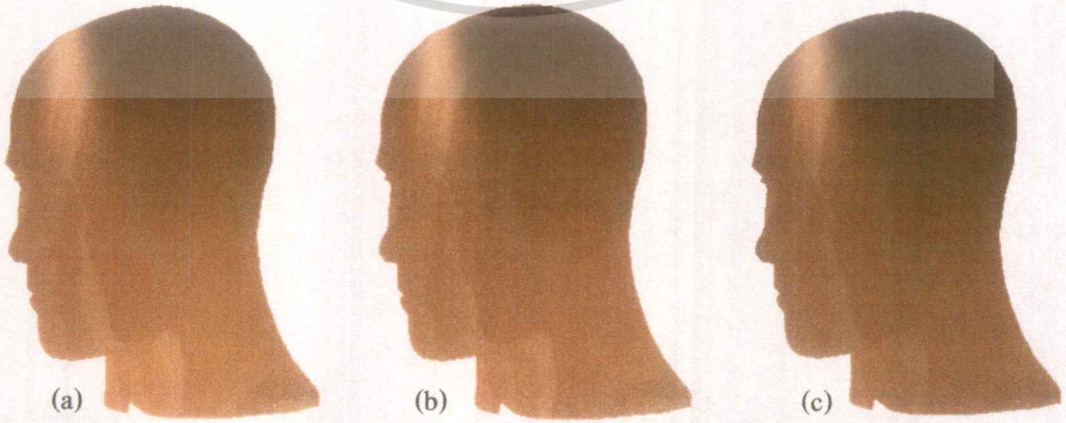


Figure 4. 18: The weathering simulation on the elephant model by the rule 25/357 (CA steps: 432, Time for complete rendering: less than 10 minutes)



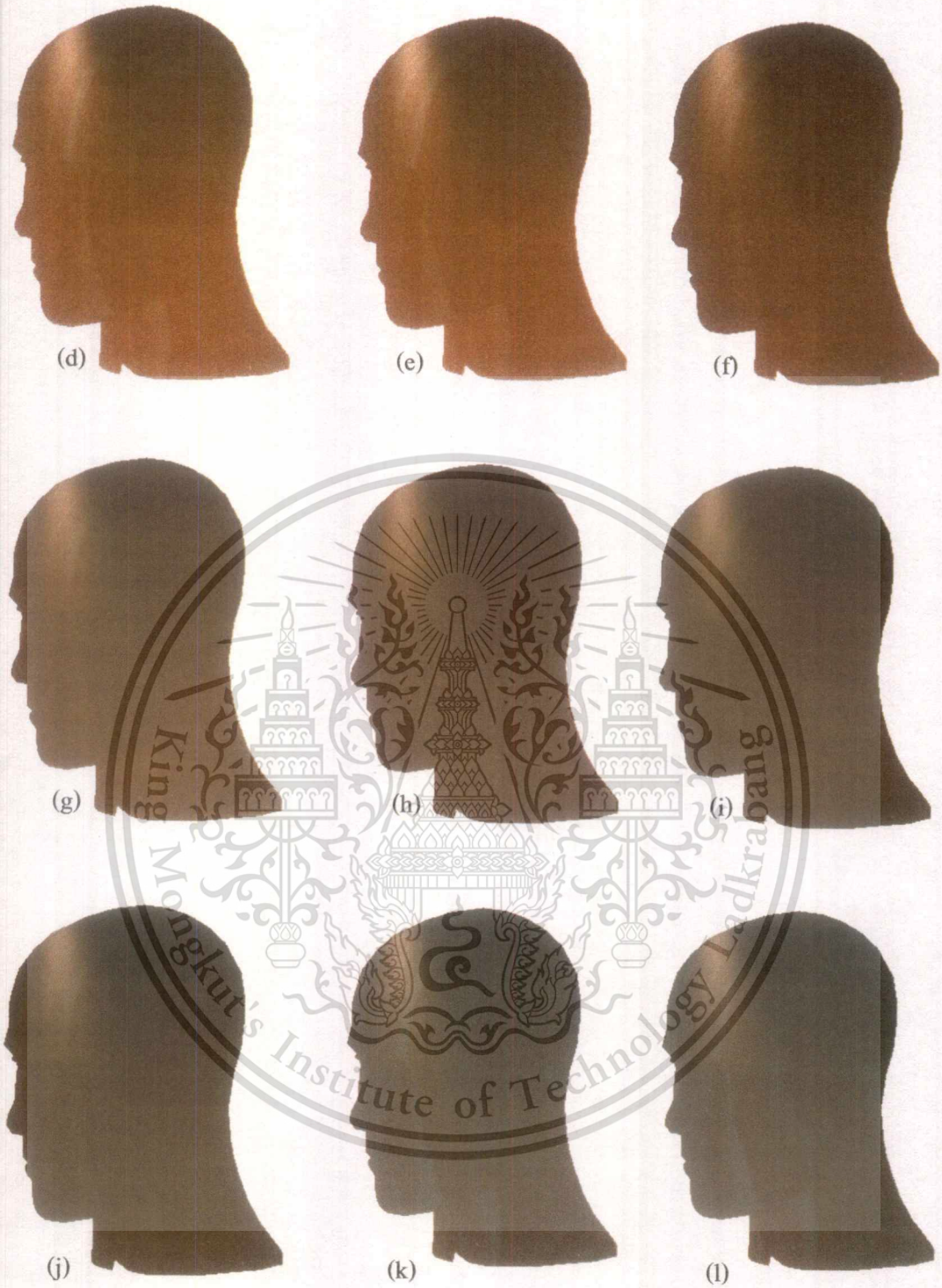


Figure 4. 19: The weathering simulation on the head model by the rule 25/357

(CA steps: 432, Time for complete rendering: less than 6 minutes)

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This research proposed the technique to simulate the phenomena of weathered wood which focuses on the change in the color of the wooden objects. This research is the computer graphics field, so realistic result is the main purpose. The combination of 3DSCA method and texture mapping is the key for realistic outcome. The expansion of the change in the color of the wooden objects is simulated by using the 3DSCA method, and the realistic results are improved by texture mapping technique. The evolution of the weathering on the wooden objects is animatedly presented.

The chapter 4 obviously shows that this simulation can not be realistic if only one of both the 3DSCA method and the texture mapping technique is used for this simulation. For example, if only the texture mapping technique is used, the result is only the pattern of wood, and any effects of wooden objects can not be generated as shown in 4.12, 4.13 and 4.14. In contrast, if the 3DSCA method is used, the result is only the change in color of the wood. It is hard to apply the CA's rule that can generate the pattern of wood and spreading the change in the color of wood simultaneously. The figures 4.8, 4.9, 4.10 and 4.11 show the results of applying only the 3DSCA method. When the advantages of these two techniques are combined, it is to improve the weathering simulation on wood more realistically as shown in figure 4.16, 4.17, 4.18 and 4.19.

5.2 Suggestions and future work

This research can apply to any 3D model, the time consumed for each process depends on the size of the 3D models. To get better results, the 3D models should not be on too big of a scale. To scale down the size of the 3D models, one can use 3Ds Max program or many 3D computer graphics programs.

The weathering simulation on wood in this research can be developed to become more realistic by adding more natural phenomena, such as: the growing and spreading of lichen, cracking of wood, etc. To add more natural phenomena, new rules of cellular automata have to be created, thus adding another possible method to this research.

REFERENCES:

[1]. A. Martinet et al, procedural Modeling of Cracks and Fractures. Shape Modeling international (Short paper), page 346-349, 2004.

[2]. B.Desbenoit et al, Simulating and modeling lichen growth. In: EUROGRAPHICS'05 Conf. Proc., Vol. 23(3), pp.341-350, 2004.

[3]. D. Foley and A.Vandama, "Fundamentals of Interactive Computer Graphics", Addison-Wesley Publishing Company, (1984): 575-587.

[4]. E. Paquette, P. Poulin, and G. Drettakis: The simulation of paint cracking and peeling. In Graphics Interface 2002, pp. 59-68, 2002.

[5]. E. Paquette, P. Poulin, Z. Drettakis Surface Aging by Impacts. Proceedings of Graphics Interface 2001 June.

[6]. E. Praun, A. Finkelstein, H. Hoppe. Lapped Textures. Siggraph 2000, Computer Graphics Proceedings.

[7]. J. Dosey and P. Hanrahan, Modeling and rendering of metallic patinas. In: SIGGRAPH'96 Conf. proc., pp.387-396, 1996.

[8]. J. Dorsey et al, Modeling and Rendering of Weathered Stone. In Proceedings of SIGGRAPH'99, pages 225-234, August 1999.

[9]. J. Neider, T. Davis and M. Woo, OpenGL Programming Guide, The Official Guide to learning OpenGLTM. SGI, Addison-Wesley Publishing Company, (1993).

[10].J. Lengyel et al, Real-time fur over arbitrary surfaces. Symposium on Interactive 3D Graphics, Pages: 227 – 232, 2001.

[11].K. Zues, R. Raum, Braunschweig: Friedrich Viewing & Sohn Verlag, 1969 [English translation: Calculating Space, MIT Technical Translation AZT-70-164-GEMIT,, MIT (Proj. MAC), Cambridge, MA 02139, Feb. 1970].

[12].N. Ganguly, B.K Sikdar et al. A survey on cellular automata. Technical report, Centre for High Performance Computing, Dresden University of Technology, December 2003.

[13].O'Brien JF and Hodgins JK, Graphical modeling and animation of brittle fracture. Proceedings of the SIGGRAPH Conference, pp 137–146, 1999.

[14].P. Sarkar, A brief history of cellular automata, ACM Computing Surveys (CSUR), Volume 32, Issue 1 (March 2000) Pages: 80 – 107, 2000.

[15].P. Even and S. Gobron. Interactive three-dimensional reconstruction and weathering simulation on buildings. CIPA 2005, XX INternational Symposium Torino, International Cooperation to Save the World's Cultural HeritageTorino, Italy, 26 September, 1 October 2005.

[16].P. Even and S. Gobron, 3D surface cellular automata and their Applications, The Journal of Visualization and Computer Animation, Volume 10, Issue 3, Page 143-158

[17].S. Gobron and N. Chiba, Crack pattern simulation based on 3D surface cellular automata. The Visual computer 17(5), pp. 287-309, 2001a.

[18].S. Gobron and N. Chiba. Simulation of peeling using 3D- Surface Cellular Automata. Proceedings of Pacific Graphics, pp.338-347, 2001.

[19].S. Merillou , J.M. Dischler , D. Ghazanfarpour. Corrosion: simulating and rendering, No description on Graphics interface 2001. Pages: 167 – 174, 2001.

[20].S. Drave et al, OpenGL programming Guide. Publisher: Addison Wesley Professional, Copyright: 2006, 896 pp

[21].X. Yin, T. Fujimoto and N. Chiba, CG Representation of Wood Aging with Distortion, cracking and Erosion. The Journal of the Society for Art and Science, Vol.3, No.4, pp.216-223, 2004.

[22]. http://www.cs.bgu.ac.il/~sipper/ca.html

[23].http://shakti.trincoll.edu/~pkenned3/alife.html

[24].http://homepages.feis.herts.ac.uk/~comqcln//CM/ca.html

[25].http://mathworld.wolfram.com/CellularAutomaton.html

[26].http://www2.bc.cc.ca.us/resperic/ca/

[27].http://cell-auto.com/neighbourhood/index.html

[28].http://mathworld.wolfram.com/MooreNeighborhood.html

[29]. http://mathworld.wolfram.com/vonNeumannNeighborhood.html

[30].http://psoup.math.wisc.edu/mcell/ca_rules.html

[31].http://student.kuleuven.be/~m0216922/CG/perlinnoise.html

[32].http://mathworld.wolfram.com/search/?q=Universal+Cellular+Automata

[33].http://mathworld.wolfram.com/Life.html

[34].http://graphics.stanford.edu/data/3Dscanrep/

[35].http://graphics.im.ntu.edu.tw/~robin/courses/cg03/model/

[36]. http://www.its.caltech.edu/~matthewf/Links.html

[37]. http://www.cc.gatech.edu/projects/large_models/index.html

[38]. http://www-static.cc.gatech.edu/~turk/bunny/bunny.html

BIOGRAPHY

PERSONAL INFORMATION:

NAME	: Lathsamy CHIDTAVONG
NATIONALITY	: Lao
DATE OF BIRTH	: 17/01/1978
ADDRESS	: COMPUTER SCIENCE DEPARTMENT, FACULTY OF SCIENCE, NATIONAL UNIVERSITY OF LAOS
DEPARTMENT	: MATHEMATICS AND COMPUTER SCIENCE

EDUCATION

1995- 1998	: Bachelor in Education (Mathematics), faculty of Education, National University of Laos, Vientiane, Lao P.D.R.

WORKING EXPERIENCE:

1998 - 2004	: work at Mathematics and Computer Science Department, Faculty of Science, National University of Laos.