國立交通大學

資訊工程系

碩 士 論 文

在可重組系統中使用動態重組排程方式增加 3D 顯像程式的效能

Using a Run-time Reconfiguration Scheduling to improve performance for 3D Rendering on Reconfigurable System

研 究 生：李孟道

指導教授：鍾崇斌 教授

中 華 民 國 九 十三 年 六 月

在可重組系統中使用動態重組排程方式增加3D顯像程式的效能

Using a Run-time Reconfiguration Scheduling to improve performance for 3D Rendering on Reconfigurable System

研 究 生：李孟道　　　　Student：Meng-Tao Lee

指導教授：鍾崇斌　　　　Advisor：Chung-Ping Chung

國 立 交 通 大 學

資 訊 工 程 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年六月

在可重組系統中使用動態重組排程方式增加3D顯像程式的效能

學生: 李孟道　　　　　　　　　　指導教授: 鍾崇斌　博士

國立交通大學資訊工程學系（研究所）碩士班

摘　要

3D 顯像程式需要大量的數學運算且資料之間的平行性極高，但是每筆資料處理的時間卻不相同，使用微處理器來執行會使得效能不夠快，設計一個特定的硬體雖然可以得到最大的效能，但是會用上非常多的硬體資源。在某些有硬體資源限制的情形下，有可能會造成效能不彰亦或是根本無法在上面執行。而可重組式運算可以在有限的硬體資源下進行大規模的運算，更能使硬體結構直接切合運算的需求，達到高效能的目的。因此在本篇論文中，探討 3D 顯像程式如何在可重組式系統上執行，透過

1. 程式分割：不破壞最大平行性的情形下，將程式分割成幾個片段
2. 工作排程：利用資料平行性盡量重用硬體和並且減少重組的次數進而降低總執行時間

透過以上兩點，我希望可以找到一個讓 3D 顯像程式在可重組式系統上執行最適合的方式。

Using a run-time Reconfiguration Scheduling to improve performance for 3D Rendering on Reconfigurable System

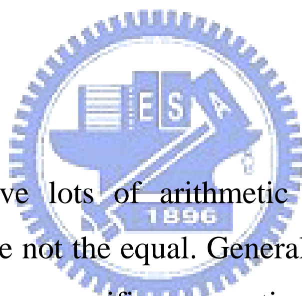
Student : Meng-Tao Lee　　　Advisors：Dr. Chung-Ping Chung

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

ABSTRACT



3D rendering applications have lots of arithmetic operations and high parallelism but execution times of each data are not the equal. General purpose computing is not fast enough for 3D rendering, Application specific computing (ASC) supports customization of applications in the form of hardware. Due to customization of hardware, this approach offers maximum performance for executing applications, but it will also cost a lot of hardware resource. In some situation with hardware constraint, it may not get high performance or even can not execute by using ASC. Reconfigurable computing can execute lots of computation with limited hardware resource and customize hardware circuit to fit the application needs for high performance.

In this thesis, I discuss how 3D rendering application can be mapped into reconfigurable system, by:

1. Analysis of rendering process: analyze operations, parallelism and computation flow
2. DFG scheduling: reuse hardware and decrease reconfigurable frequency using data parallelism to minimize total execution time

Through this, I propose a method to execute 3D rendering application on reconfigurable system.

誌謝

首先感謝我的指導老師 鍾崇斌教授，在他的諄諄教誨、辛勤指導與勉勵下，得以順利完成此篇論文。同時感謝我的口試委員陳添福教授以及單智君教授，在他們的建議之下，使此篇論文更加完整。

感謝 Reconfigurable Computing 研究群的博士班學長—蔣坤成學長，以及其他研究團體的博士班學長們—鄭哲聖學長、喬偉豪學長和林漢君學長。也感謝實驗室其他同學們熱心的與我討論，給我意見和鼓勵。

此外，感謝諸位同學和學弟妹們，你們的陪伴讓我的生活充滿歡樂；也讓這兩年來的研究生活更加的多采多姿與充實。最後感謝我的家人，謝謝你們在背後全心全意的支持我、關懷我與鼓勵我。讓我在這研究的路上走的更順利，進而能更全無後顧之憂的用功學習，讓我能堅持追其自己的理想。

所有支持我、勉勵我的師長與親友，奉上我最誠摯的感謝與祝福，謝謝你們。

李孟道
2004.6.28

Contents

摘　要 i
ABSTRACT ii
誌謝 iii
Contents iv
List of Tables vi
List of Figures vii
Chapter 1　Introduction 1
1.1　Computer Graphics 1
1.2　Run-time Reconfiguration 2
1.3　Motivation and Objective 4
1.4　Organization of This Thesis 4
Chapter 2　Background 5
2.1　Reconfigurable Models 5
2.1.1　Single Context 5
2.1.2　Multicontext 6
2.1.3　Partially Reconfigurable 7
2.2　Rendering Process 8
2.2.1　Transformation 9
2.2.2　Lighting 13
2.2.3　Projection 23
2.2.4　Rasterization 25
2.2.5　Shading 31
Chapter 3　Design 35
3.1　Analysis 35
3.2　Design 39
3.3　Observation 40
3.4　Opportunity of improvement 40
3.5　Dynamic Task Scheduling 41
Chapter 4　Simulation 42

4.1 Simulation Environment 42
4.2 Simulation Result 42
Chapter 5 Conclusion and Future work 47
5.1 Conclusion 47
5.2 Future work 47
Reference 49

List of Tables

Table 3-1 Basic operation for each stage 38
Table 3-2 Resource and execution time of each stage 39
Table 4-1 Benchmarks 43
Table 4-2 Execution time comparison 1 43
Table 4-3 execution time comparison 2 44
Table 4-4 The comparison of fixed configuration design and RTR 46

List of Figures

Figure 1-1 The concept of Run-time Reconfiguration 2
Figure 2-1 Reconfigurable Models 6
Figure 2-2 The architecture of programming bit 6
Figure 2-3 The minimum entities required in a practical viewing system 11
Figure 2-4 The calculation of up vector V 13
Figure 2-5 The difference between local reflection models and shading algorithms 14
Figure 2-6 The concept of local reflection model 15
Figure 2-7 The reflection phenomena 16
Figure 2-8 The computer graphics surface 16
Figure 2-9 The Phong diffuse component 19
Figure 2-10 The Phong specular component 20
Figure 2-11 The light intensity 20
Figure 2-12 the vector H 21
Figure 2-13 The orientation of the light source 22
Figure 2-14 The projection in computer graphics 23
Figure 2-15 The perspective projection 24
Figure 2-16 Deriving a perspective transformation 24
Figure 2-17 The concept of Bresenham's algorithm 26
Figure 2-18 The linked list 29
Figure 2-19 The problem with polygon boundaries 30
Figure 2-20 The result of Rasterization rules 31
Figure 2-21 The Gouraud Shading 32
Figure 2-22 The difference between Gouraud and Phong shading 34
Figure 3-1 The example of Rendering process 35
Figure 3-2 The DFG of one vertex 35
Figure 3-3 The execution example of fixed configuration design 40
Figure 3-4 The execution example of modified fixed configuration design 40
Figure 3-5 The execution example of Run-time Reconfiguration design 41

Figure 4-1 The simulation result of different PE..45
Figure 4-2 The simulation result of cost function 45

Chapter 1 Introduction

Computer graphics are traditionally computation intensive, it stared at high-end system such as work stations for scientific purpose and then it showed up at the desktop computer. For now, portable devices are going to have ability to perform 3D applications. As we can see, the hardware resources for the computer graphics are getting less and less. If we can use lesser hardware to accelerated larger portions of application, it will be beneficial for the small system. In this thesis, we propose a method to analyze the application and propose a run-time scheduling for that reconfigurable system.

1.1 Computer Graphics

Since the mid-1970s the developmental motivation of computer graphics from the viewpoint of its practitioners has been photorealism or the pursuit of techniques that make a graphics image of an object or scene in distinguishable from a TV image or photograph. A more recent strand of the application of these techniques is to display information in, for example, medicine, science and engineering.

The calculation of light-object interaction is the foundation of photo-realism and this split neatly into two fields – the development of local reflection models and the development of global models. Local or direct reflection models only consider the interaction of an object with a light source as if the object and light were floating in dark space. That is, only the first reflection of light from the object is considered. Global reflection models consider how light reflects from one object and travels onto another. In other words the light impinging on a point on the surface can come either from a light source (direct light) or indirect light that has first hit another object. Although two partial solutions for global interaction, ray tracing and radiosity, are implementer, global interaction is still for the most part an unsolved problem.

Much modern scientific research comes from computer graphics research and early

major advances are created and consolidated into a practical technology. Later significant advances seem to be more difficult to achieve. We can say that most images are produced using the Phong local reflection model (first reported in 1975), fewer using ray tracing (first popularized in 1980) and fewer still using radiosity (first reported in 1984). Although there is still much research being carried out in light-scene interaction methodologies much of the current research in computer graphics is concerned more with applications, for example, with such general applications as animation, visualization and virtual reality. In the most important computer graphics publication (the annual SIGGRAPH conference proceedings) there was in 1985 a total of 22 papers concerned with the production techniques of images (rendering, modeling and hardware) compared with 13 on what could loosely be called applications. A decade later in 1995 there were 37 papers on applications and 19 on image production techniques. [1]

1.2 Run-time Reconfiguration

Frequently, the areas of a program that can be accelerated by using the reconfigurable hardware are too numerous or complex to be loaded simultaneously onto the available hardware. For these cases, if we can swap different configurations in and out of the reconfigurable hardware as they are needed during program execution, it will be beneficial (Figure 1.1). This concept is called run-time reconfiguration (RTR).

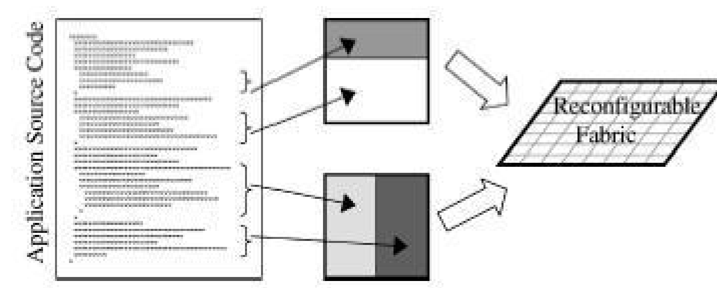


Figure 1-1 The concept of Run-time Reconfiguration [2]

Run-time reconfiguration is based upon the concept of virtual hardware, which is similar to virtual memory. Here, the physical hardware is much smaller than the sum of the resources required by each of the configurations. Therefore, instead of reducing the

number of configurations that are mapped, we instead swap them in and out of the actual hardware as they are needed. Because run-time reconfiguration allows more sections of an application to be mapped into hardware than can be fit in a non-run-time reconfigurable system, a greater portion of the program can be accelerated. This provides potential for an overall improvement in performance.

Configurations are swapped in and out of the reconfigurable hardware during a single program's execution. Some of these configurations will likely require access to the results of other configurations. Configurations that are active at different periods in time therefore must be provided with a method to communicate with one another. Primarily, this can be done through the use of registers [7] [8] [9] [10], the contents of which can remain intact between reconfigurations. This allows one configuration to store a value, and a later configuration to read back that value for use in further computations. An alternative for reconfigurable systems that do not include state-holding devices is to write the result back to registers or memory external to the reconfigurable array, which is then read back by successive configurations [11].

There are a few different configuration memory styles that can be used with reconfigurable systems. A single context device is a serially programmed chip that requires a complete reconfiguration in order to change any of the programming bits. A multicontext device has multiple layers of programming bits, each of which can be active at a different point in time. Devices that can be selectively programmed without a complete reconfiguration are called partially reconfigurable. These different types of configuration memory are described in more detail later. An advantage of the multicontext FPGA over a single context architecture is that it allows for an extremely fast context switch (on the order of nanoseconds), whereas the single context may take milliseconds or more to reprogram. The partially reconfigurable architecture is also more suited to run-time reconfiguration than the single context, because small areas of the array can be modified without requiring that the entire logic array be reprogrammed.

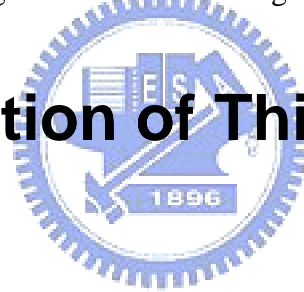
For all of these run-time reconfigurable architectures, there are also some compilation issues which are not encountered in systems that only configure at the beginning of an application. For example, run-time reconfigurable systems are able to optimize based on values that are known only at run-time. Furthermore, compilers must consider the run-time reconfigurability when generating the different circuit mappings, not only to be aware of the increase in time-multiplexed capacity, but also to schedule reconfigurations so as to minimize the overhead that they incur. These software issues, as well as an

overview of methods to perform fast configuration, will be explored in the sections that follow.

1.3 Motivation and Objective

Now graphics accelerators are usually made by dedicated hardware, it can achieve very high performance but it loses the flexibility. Reconfigurable computing provides another way to accelerate the computer graphics applications which retain flexibility of a software solution. Once if we use reconfigurable hardware to accelerate such a application, scheduling is a critical issue that execution time of each stage in rendering process is not constant.

Here our objective is to analyze operations, parallelism and computation flow of rendering process to design a Run-time reconfiguration scheduling.


1896

1.4 Organization of This Thesis

The organization of this thesis is as follows: In Chapter 2, the background is presented. In Chapter 3, analysis of the rendering process and the design of Run-time reconfiguration are described. In Chapter 4, we analyze our simulation result and show our simulation environment. Finally, conclusion and future work are presented in Chapter 5.

Chapter 2 Background

In this chapter, we will introduce the backgrounds of reconfigurable models and computer graphics.

2.1 Reconfigurable Models

Traditional FPGA structures have been single context which only allow one full-chip configuration to be loaded at a time. However, designers of reconfigurable systems have found this style of configuration to be too limiting or slow to efficiently implement run-time reconfiguration. The following discussion (reference from [2]) defines the single context device, and further considers newer FPGA designs (multicontext and partially reconfigurable), along with their impact on run-time reconfiguration.

2.1.1 Single Context

Current single context FPGAs are programmed using a serial stream of configuration information. Because only sequential access is supported, any change to a configuration on this type of FPGA requires a complete reprogramming of the entire chip. Although this does simplify the reconfiguration hardware, it does incur a high overhead when only a small part of the configuration memory needs to be changed. Many commercial FPGAs are of this style, including the Xilinx 4000 series [12], the Altera Flex10K series [13], and Lucent's Orca series [14]. This type of FPGA is therefore more suited for applications that can benefit from reconfigurable computing without run-time reconfiguration. A single context FPGA is depicted in Figure 2.1.

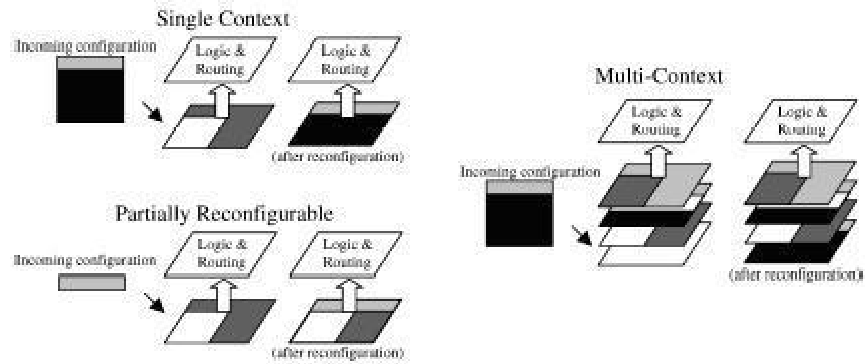


Figure 2-1 Reconfigurable Models [2]

In order to implement run-time recon-figuration onto a single context FPGA, the configurations must be grouped into contexts, and each full context is swapped in and out of the FPGA as needed. Because each of these swap operations involve reconfiguring the entire FPGA, a good partitioning of the configurations between contexts is essential in order to minimize the total reconfiguration delay. If all the configurations used within a certain time period are present in the same context, no reconfiguration will be necessary. However, if a number of successive configurations are each partitioned into different contexts, several reconfigurations will be needed, slowing the operation of the runtime reconfigurable system.

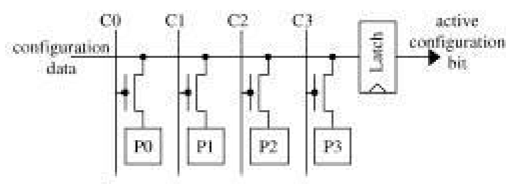


Figure 2-2 The architecture of programming bit [2]

A four-bit multicontexted programming bit[16]. P0-P3 are the stored programming bits, while C0-C3 are the chip wide control lines that select the context to program or activate.

2.1.2 Multicontext

A multicontext FPGA includes multiple memory bits for each programming bit location [10] [15] [16] [17]. These memory bits can be thought of as multiple planes of configuration information, as shown in Figure 2.2. One plane of configuration information can be active at a given moment, but the device can

quickly switch between different planes, or contexts, of already-programmed configurations. In this manner, the multicontext device can be considered a multiplexed set of single context devices, which requires that a context be fully reprogrammed to perform any modification. This system does allow for the background loading of a context, where one plane is active and in execution while an inactive place is in the process of being programmed. Figure 2.2 shows a multicontext memory bit, as used in [16]. A commercial product that uses this technique is the CS2000 RCP series from Chameleon, Inc [17]. This device provides two separate planes of programming information. At any given time, one of these planes is controlling current execution on the reconfigurable fabric, and the other plane is available for background loading of the next needed configuration.

Fast switching between contexts makes the grouping of the configurations into contexts slightly less critical, because if a configuration is on a different context than the one that is currently active, it can be activated within an order of nanoseconds, as opposed to milliseconds or longer. However, it is likely that the number of contexts within a given program is larger than the number of contexts available in the hardware. In this case, the partitioning again becomes important to ensure that configurations occurring in close temporal proximity are in a set of contexts that are loaded into the multicontext device at the same time. More aspects involving temporal partitioning for single- and multicontext devices will be discussed in the section on compilers for run-time reconfigurable systems.

2.1.3 Partially Reconfigurable

In some cases, configurations do not occupy the full reconfigurable hardware, or only a part of a configuration requires modification. In both of these situations, a partial reconfiguration of the array is required, rather than the full reconfiguration required by a single- or multicontext device. In a partially reconfigurable FPGA, the underlying programming bit layer operates like a RAM device. Using addresses to specify the target location of the configuration data allows for selective reconfiguration of the array. Frequently, the undisturbed portions of the array may continue execution, allowing the overlap of computation with

reconfiguration. This has the benefit of potentially hiding some of the reconfiguration latency.

When configurations do not require the entire area available within the array, a number of different configurations may be loaded into unused areas of the hardware at different times. Since only part of the array is reconfigured at a given point in time, the entire array does not require reprogramming. Additionally, some applications require the updating of only a portion of a mapped circuit, while the rest should remain intact, as shown in Figure 2.1. For example, in a filtering operation in signal processing, a set of constant values that change slowly over time may be reinitialized to a new value, yet the overall computation in the circuit remains static. Using this selective reconfiguration can greatly reduce the amount of configuration data that must be transferred to the FPGA. Several run-time reconfigurable systems are based upon a partially reconfigurable design, including Chimaera [18], PipeRench [8] [19], NAPA [9], and the Xilinx 6200 and Vertex FPGAs [20] [21].

Unfortunately, since address information must be supplied with configuration data, the total amount of information transferred to the reconfigurable hardware may be greater than what is required with a single context design. This makes a full reconfiguration of the entire array slower than the single context version. However, a partially reconfigurable design is intended for applications in which the size of the configurations is small enough that more than one can fit on the available hardware simultaneously. Plus, as we discuss in subsequent sections, a number of fast configuration methods have been explored for partially reconfigurable systems in order to help reduce the configuration data traffic requirements.

2.2 Rendering Process

In this thesis, we study the basic component of computer graphics from Alan Watt's book [1]. Hence the following background is to refer to Alan Watt's book [1].

2.2.1 Transformation

Transformation includes three coordinate spaces:

A. Local or modeling coordinates systems

For ease of modelling it makes sense to store the vertices of a polygon mesh object with respect to some point located in or near the object. For example, we would almost certainly want to locate the origin of a cube at one of the cube vertices, or we would want to make the axis of symmetry of an object generated as a solid of revolution, coincident with the z axis. As well as storing the polygon vertices in a coordinate system that is local to the object, we would also store the polygon normal and the vertex normals. When local transformations are applied to the vertices of an object, the corresponding transformations are applied to the associated normals.

B. World coordinate systems

Once an object has been modeled the next stage is to place it in the scene that we wish to render. All objects that together constitute a scene have their separate local coordinate systems. The global coordinate system of the scene is known as the 'world coordinate system'. All objects have to be transformed into this common space in order that their relative spatial relationships may be defined. The act of placing an object in a scene defines the transformation required to take the object from local space to world space. If the object is being animated, then the animation system provides a time-varying transformation that takes the object into world space on a frame by frame basis.

The scene is lit in world space. Light sources are specified, and if the shaders within the renderer function are in world space then this is the final transformation that the normals of the object have to undergo. The surface attributes of an object – texture, colour, and so on – are specified and tuned in this space.

C. Camera or eye view coordinate system

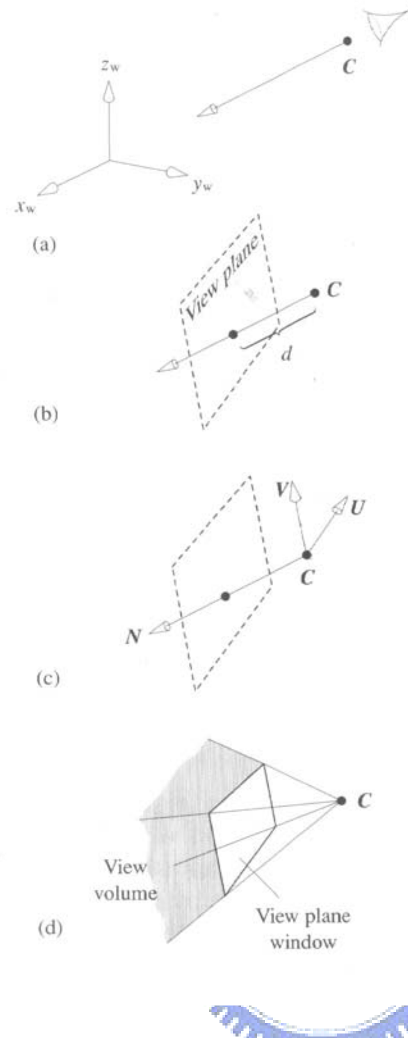
The eye, camera or view coordinate system is a space that is used to establish

viewing parameters (view point, view direction) and a view volume. (A virtual camera is often used as the analogy in viewing systems, but if such an allusion is made we must be careful to distinguish between external camera parameters or those that affect the nature and size of the image on the film plane. Most rendering systems imitate a camera which in practice would be a perfect pinhole. However, there are other facilities in computer graphics that cannot be imitated by a camera and because of this the analogy is of limited utility.)

We will now deal with a basic view coordinate system and the transformation from world space to view coordinate space. The reason that this space exist, after all we could go directly from world space to screen space, is that certain operations (and specifications) are most conveniently carried out in view space.

We define a viewing system as being the combination of a view coordinate system together with the specification of certain facilities such as a view volume. The simplest or minimum system would consist of the following：

- A view point which establishes the viewer's position in world space; this can either be the origin of the view coordinate system or the centre of projection together with a view direction ***N***.
- A view coordinate system defined with respect to the view point.
- A view plane onto which the two-dimensional image of the scene is projected.
- A view frustum or volume which defines the field or view.



The minimum entities required in a practical viewing system.

(a) View point **C** and viewing direction **N**.
(b) A view plane normal to the viewing direction **N** positioned *d* units from **C**.
(c) A view coordinates system with the origin **C** and **UV** axes embedded in plane parallel to the view plane.
(d) A view volume defined by the frustum formed by **C** and the view plane window.

Figure 2-3 The minimum entities required in a practical viewing system [1]

These entities are show in Figure 2.3. The view coordinate system, ***UVN***, has ***N*** coincident with the viewing direction and ***V*** and ***U*** lying in a plane parallel to the view plane. We can consider the origin of the system to be the vie point ***C***. The view plane containing ***U*** and ***V*** is of infinite extent and we specify a view volume or frustum which defines a window in the view plane. It is the contents of this window – the projection of that part of the scene that is contained within the view volume – that finally appears in the screen.

Thus, using the virtual camera analogue we have a camera that can be positioned anywhere in world coordinate space, pointed in any direction and rotated about the viewing direction ***N***.

To transform points in world coordinate space we invoke a change of coordinate system transformation and this split into two components: a translational one and a rotational one. Thus:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = T_{view} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \quad (1)$$

where:

$$T_{view} = RT \quad (2)$$

and:

$$T = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3)$$

The only problem now is specifying a user interface for the system and mapping whatever parameters are used by the interface into ***U***, ***V*** and ***N***. A user needs to specify ***C***, ***N*** and ***V***. ***C*** is easy enough. ***N***, the viewing direction or view plane normal, can be entered say, using two angles in a spherical coordinate system – this seems reasonably intuitive:

θ the azimuth angle

ϕ the colatitude or elevation angle

where:

$$\begin{aligned} N_x &= \sin\phi\cos\theta \\ N_y &= \sin\phi\sin\theta \\ N_z &= \cos\phi \end{aligned} \quad (4)$$

V is more problematic. For example, a user may require 'up' to be the same sense as 'up' in the world coordinates system. However, this cannot be achieved by setting:

$$V = (0,0,1)$$

because ***V*** must be perpendicular to ***N***. A sensible strategy is to allow a user to specify an approximate orientation for ***V***, say ***V'*** and have the system calculate ***V***. Figure 2.4 demonstrate this. ***V'*** is the user-specified up vector. This is projected onto the view plane:

$$V = V' - (V' \bullet N)N \quad (5)$$

and normalized. U can be specified or not depending on the user's requirements. If U is unspecified, it is obtained from:

$$U = N \times V \quad (6)$$

This results in a left-hand coordinate system, which although somewhat inconsistent, conforms with our intuition of a practical viewing system, which has increasing distances from the view point as increasing values along the view direction axis. Having established the viewing transformation using ***UVN*** notation, we will in subsequent section use (x_v, y_v, z_v) to specify points in the view coordinate system.

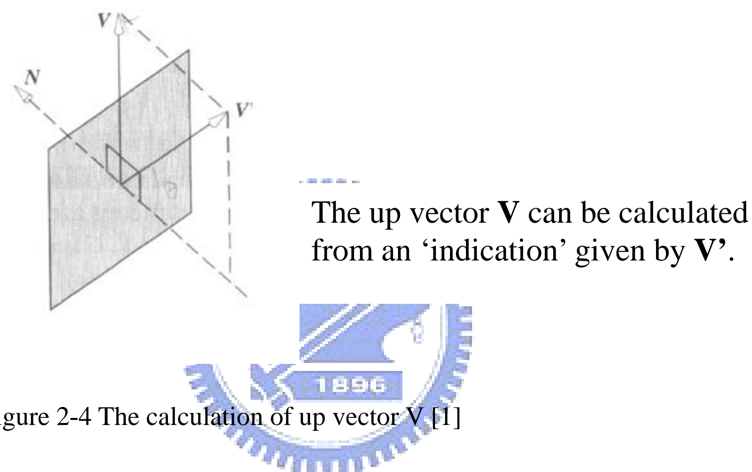


Figure 2-4 The calculation of up vector V [1]

2.2.2 Lighting

Shading Pixels

The first quality shading in computer graphics was developed by H. Gouraud in 1971 (Gouraud 1971). In 1975 Phong Bui-Tuong (Phong 1975) improved on Gouraud's model and Phong shading, as it is universally known, became the *defacto* standard in mainstream 3D graphics. Despite the subsequent development of 'global' techniques, such as ray tracing and radiosity, Phong shading has remained ubiquitous. This is because it enables reality to be mimicked to an acceptable level at reasonable cost.

There are two separate considerations to shading the pixels onto which a polygon projects. First we consider how to calculate the light reflected at any point

on the surface of an object. Given a theoretical framework that enables us to do this, we can then calculate the light intensity at the pixels onto which the polygon projects. The first consideration we call 'local reflection models' and the second 'shading algorithms'. The difference is illustrated conceptually in Figure 2.5. For example, one of the easiest approaches to shading – Gouraud shading – applies a local reflection models at each of the vertices to calculate vertex intensity, then derives a pixel intensity using the interpolation equations.

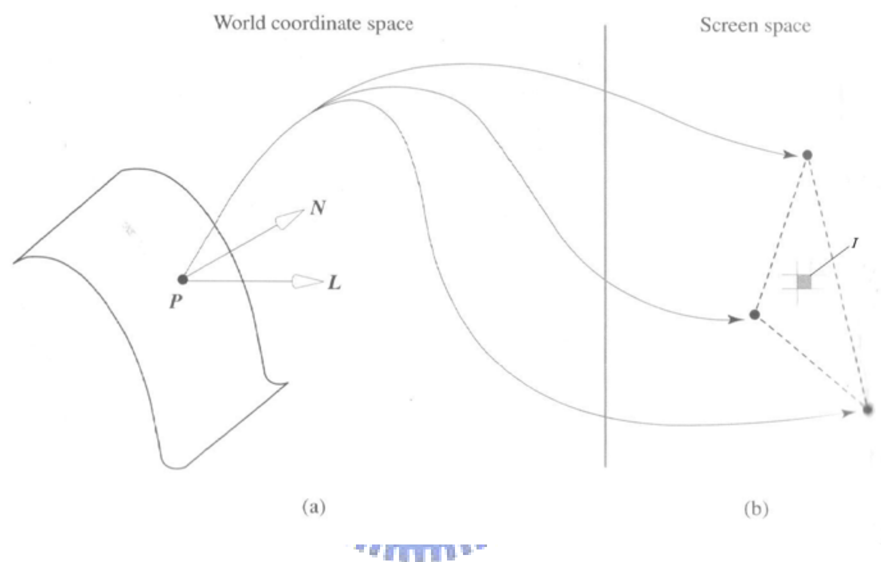


Figure 2-5 The difference between local reflection models and shading algorithms [1]

Illustrating the difference between local reflection models and shading algorithms.

(a) Local reflection models calculate light intensity as any point **P** on the surface of an object.
(b) Shading algorithms interpolate pixel values from calculated light intensities at the polygon vertices.

Basically there is a conflict here. We only want to calculate the shade for each pixel onto which the polygon projects. But the reflected light intensity at every point on the surface of a polygon is by definition a world space calculation. We are basing the calculation on the orientation of the surface with respect to a light source both of which are defined in world space. Thus we use a 2D projection of the polygon as the basis of an interpolation scheme that controls the world space calculations of intensity and this is incorrect. Linear interpolation, using equal

increments, in screen space does not correspond to how the reflected intensity should vary across the face of the polygon in world space. One of the reasons for this is that we have already performed a (non-linear) perspective transformation to get into screen space. Like many algorithms in 3D computer graphics it produces an acceptable visual result, even using incorrect mathematics. However, this approach does lead to visible artifacts in certain contexts.

Local reflection models

A local reflection model enables the calculation of the reflected light intensity from a point on the surface of an object. Here we will confine ourselves to considering, from a practical view point, the most common model and see how it fits into a renderer.

This model, introduced in 1975, evaluates the intensity of the reflected light as a function of the orientation of the surface at the point of interest with respect to the position of a point light source and surface properties. We refer to such a model as a local reflection model because it only considers direct illumination. It is as if the object under consideration was an isolated object floating in free space. Interaction with other objects that result in shadows and inter-reflection are not taken into account by local reflection models. This point is emphasized in Figure 2.6.

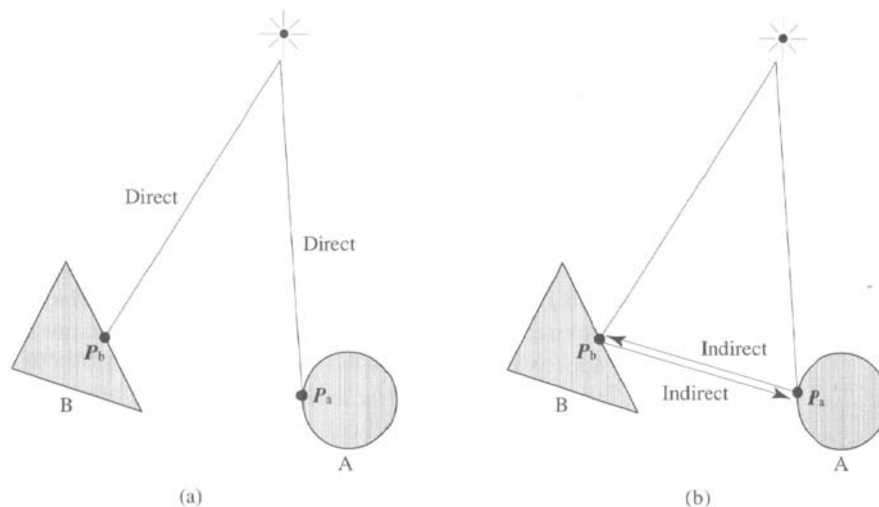


Figure 2-6 The concept of local reflection model [1]

(a) A local reflection model calculates intensity at $\mathbf{P}_b$ and $\mathbf{P}_a$ considering direct illumination only.
(b) Any indirect reflected light from A to B or from B to A is not taken into account.

The physical reflection phenomena that the model simulates are:

- Perfect specular reflection
- Imperfect specular reflection
- Perfect diffuse reflection

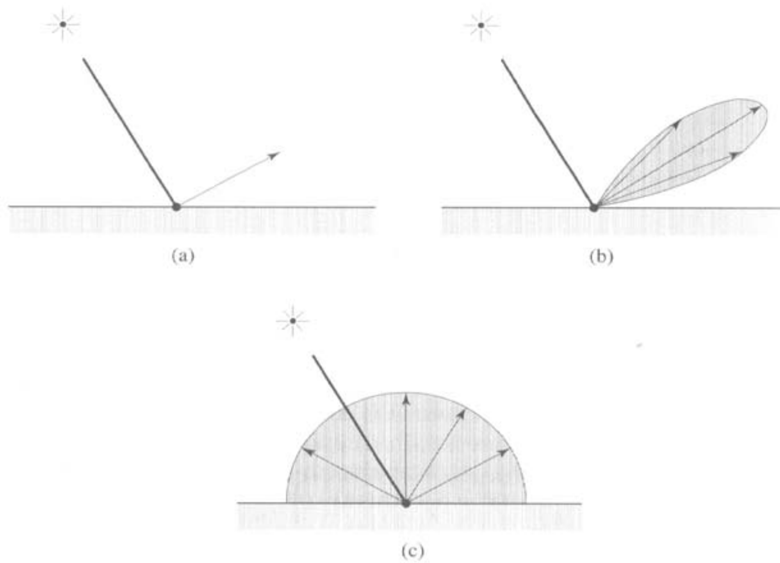


Figure 2-7 The reflection phenomena [1]

The three reflection phenomena used in computer graphics.
(a) Perfect specular reflection.
(b) Imperfect specular reflection.
(c) Perfect diffuse reflection.

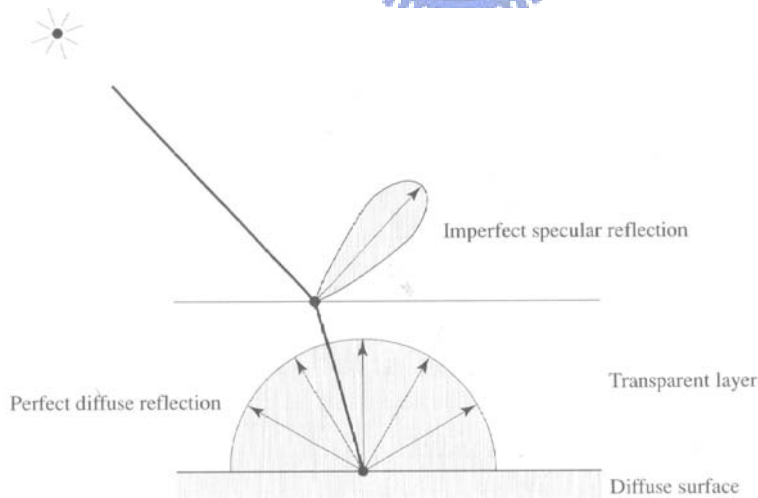


Figure 2-8 The computer graphics surface [1]

These are illustrated in Figure 2.7 for a point light source that is sending an infinitely thin beam of light to a point on a surface. Perfect specular reflection occurs when incident light is reflected, without diverging, in the 'mirror' direction.

Imperfect specular reflection is that which occurs when a thin beam of light strikes an imperfect mirror, that is a surface whose reflecting properties are that of a perfect mirror but only at a microscopic level – because the surface is physical rough. Any area element of such a surface can be considered to be made up of thousands of tiny perfect mirrors all at slightly different orientations.

Perfect specular reflection does not occur in practical but we use it in ray tracing models simply because calculating interaction due to imperfect specular reflection is too expensive. A perfect diffuse surface reflects the light equally in all directions and such a surface is usually called matte.

The Phong reflection model considers the reflection from a surface to consist of three components linearly combined:

Reflected light = ambient light + diffuse component + specular component

The ambient term is a constant and simulates global or indirect illumination. This term is necessary because parts of a surface that cannot 'see' the light source, but which can be seen by the viewer, need to be lit. Otherwise they would be rendered as black. In reality such lighting comes from global or indirect illumination and simply adding a constant side-step the complexity of indirect or global illumination calculation.

It is useful to consider what types of surface such a model simulates. Linear combination of a diffuse and specular component occurs in polished surfaces such as varnished wood. Specular reflection results from the transparent layer and diffuse reflection from the underlying surface (Figure 2.8). Many different physical types, although not physical the same as a varnished wood, can be approximately simulated by the same model. The veracity of this can be demonstrated by considering looking at a sample of real varnished wood, shiny plastic and gloss paint. If all contextual clues are removed and the reflected light from each sample exhibited the same spectral distribution, an observer would find it difficult to distinguish between the samples.

As well as possessing the limitation of being a local model, the Phong reflection model is completely empirical or imitative. One of its major defects is that the value of reflected intensity calculated but the model is a function only of the viewing direction and the orientation of the surface with respect to the light source. In practical, reflected light intensity exhibits bi-directional behavior. It depends also on the direction of the incident light. This defect has led to much research into

physically based reflection models, where an attempt is made to model reflected light by simulating real surface properties. However, the subtle improvements possible by using such models – such as the ability to make surface look metallic – have not resulted in the demise of the Phong reflection model and the main thrust of current research into rendering methods deals with the limitation of ‘localness’. Global methods, such as radiosity, result in much more significant improvements to the apparent reality of a scene.

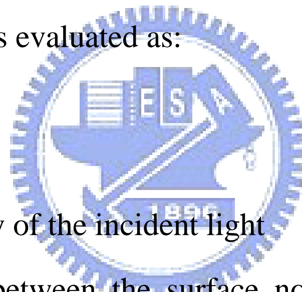
Leaving aside, for a moment, the issue of color, the physical nature of a surface is simulated by controlling the proportion of the diffuse to specular reflection and we have the reflected light:

$$I = k_a I_a + k_d I_d + k_s I_s \quad (7)$$

Where the proportions of the three components, ambient, diffuse and specular are controlled by three constant, where:

$$k_a + k_d + k_s = 1 \quad (8)$$

Consider I_d. This is evaluated as:



$$I_d = I_i \cos\theta \quad (9)$$

where:

I_i is the intensity of the incident light

θ is the angle between the surface normal at the point of interest and the direction of the light source

In vector notation:

$$I_d = I_i(\overline{L} \bullet \overline{N}) \quad (10)$$

The geometry is shown in Figure 2.9

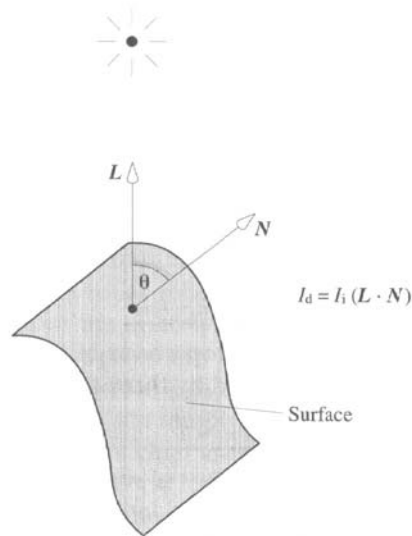


Figure 2-9 The Phong diffuse component [1]

Now physically the specular reflection consists of an image of the light source 'smeared' across an area of the surface resulting in what is commonly known as a highlight. A highlight is only seen by a viewer if the viewing direction is near to the mirror direction. We therefore simulate specular reflection by:

$$I_s = I_i \cos^n \Omega \qquad (11)$$

where:

Ω is the angle between the viewing direction and the mirror direction $\overline{R}$

n is an index that simulate the degree of imperfection of a surface

When $n = \infty$ the surface is a perfect mirror – all reflected light emerges along the mirror direction. For other values of n an imperfect specular reflector is simulated (Figure 2.7 b). The geometry of this is shown in Figure 2.10. In vector notation we have:

$$I_s = I_i (\overline{R} \bullet \overline{V})^n \qquad (12)$$

Bringing these terms together gives:

$$I = k_a I_a + I(k_d (\overline{L} \bullet \overline{N}) + k_s (\overline{R} \bullet \overline{V})^n) \qquad (13)$$

The behavior of this equation is illustrated in Figure 2.11. Figure 2.11 shows the light intensity at a single point ***P*** as a function of the orientation of the viewing vector $\overline{V}$. The semicircle is the sum of the constant ambient term and the diffuse

term – which is constant for a particular value of $\overline{N}$. Addition of the specular term gives the profile shown in the figure. As the value of n is increased the specular bump is narrowed.

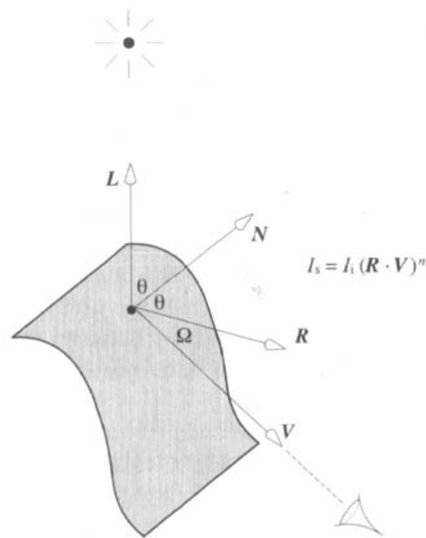


Figure 2-10 The Phong specular component [1]

E S A

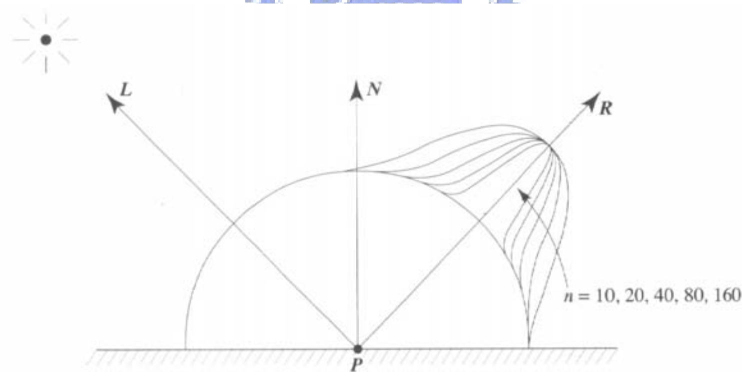


Figure 2-11 The light intensity [1]

The light intensity at point **P** as a function of the orientation of the viewing vector **V**.

Local reflection model – practical points

A number of practical matters that deal with color and the simplification of the geometry now need to be explained.

The expense of the above shading equation, which is applied a number of times at every pixel, can be considerably reduced by making geometric simplifications that

reduce the calculation time, but which do not affect the quality of the shading. First if the light source is considered as a point source located at infinity the $\overline{L}$ is constant over the domain of the scene. Second we can also place the view point at infinity making V constant. Of course, for the view and perspective transformation, the view point needs to be firmly located in world space so we end up using a finite view point for the geometric transformation and an infinite one for the shading equation.

Next the vector $\overline{R}$ is expensive to calculate and it is easier to define a vector $\overline{H}$ (Halfway) which is the unit normal to a hypothetical surface that is oriented in a direction halfway between the light direction vector $\overline{L}$ and the viewing vector $\overline{V}$ (Figure 2.12). It is easily seen that:

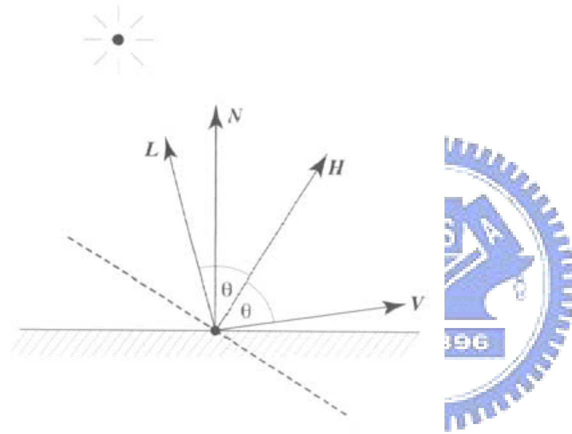


Figure 2-12 the vector H [1]

H is the normal to a surface orientation that would reflect all the light along **V**.

$$\overline{H} = (\overline{L} + \overline{V})/2 \quad (14)$$

This is the orientation that a surface would require if it was to reflect light maximally along the $\overline{V}$ direction. Our shading equation now becomes:

$$I = I_a k_a + I_i (k_d (\overline{L} \bullet \overline{N}) + k_s (\overline{N} \bullet \overline{H})^n) \quad (15)$$

because the term $(\overline{N} \bullet \overline{H})$ varies in the same manner as $(\overline{R} \bullet \overline{V})$. These simplifications mean that I is now a function only of N.

For colored objects we generate three components of the intensity I_r, I_g and I_b controlling the color of the objects by appropriate setting of the diffuse

reflection coefficients k_r, k_b and k_g. In effect the specular highlight is just the reflection of the light source in the surface of the object and we set the proportions of the k_s to match the color of the light. For a white light, k_s is equal in all three equations. Thus we have:

$$I_r = I_a k_{ar} + I_i((k_{dr}(\overline{L}\bullet\overline{N}) + k_{sr}(\overline{N}\bullet\overline{H})^n)$$
$$I_g = I_a k_{ag} + I_i((k_{dg}(\overline{L}\bullet\overline{N}) + k_{sg}(\overline{N}\bullet\overline{H})^n) \quad (16)$$
$$I_b = I_a k_{ab} + I_i((k_{db}(\overline{L}\bullet\overline{N}) + k_{sb}(\overline{N}\bullet\overline{H})^n)$$

Local reflection model – light source considerations

One of the most limiting approximations in the above model is reducing the light source to a point at infinity. A simple directional light (non-point) is easily modeled and the following was suggested by Warn (1983). In this method a directional light source is modeled in the same way as a specularly reflecting surface, where the light emitted from the source is given by a cosine function raised to a power. Here we assume that for a directional source, the light intensity in a particular direction, given by the angle ϕ is :

$$I_s \cos^m \phi \quad (17)$$

Now ϕ is the angle between $-\overline{L}$, the direction of the point on the surface that we are considering, and $\overline{L_s}$, the orientation of the light source (Figure 2.13). The value of I_i that we use in the shading equation is then given by:

$$I_i = I_s(-\overline{L}\bullet\overline{L_s})^m \quad (18)$$

Note that we can no longer consider the vector $\overline{L}$ constant over the scene.

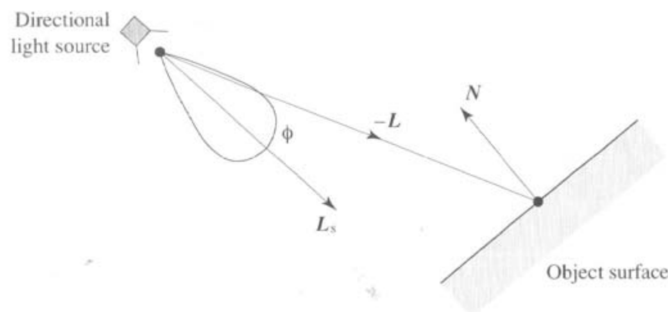


Figure 2-13 The orientation of the light source [1]

Light source represented as a specularly reflecting surface.

2.2.3 Projection

Because the viewing surface in computer graphics is deemed to be flat we consider the class of projections known as planar geometric projections. Two basic projections, perspective and parallel, are now described. These projections and the difference in their nature is illustrated in Figure 2.14.

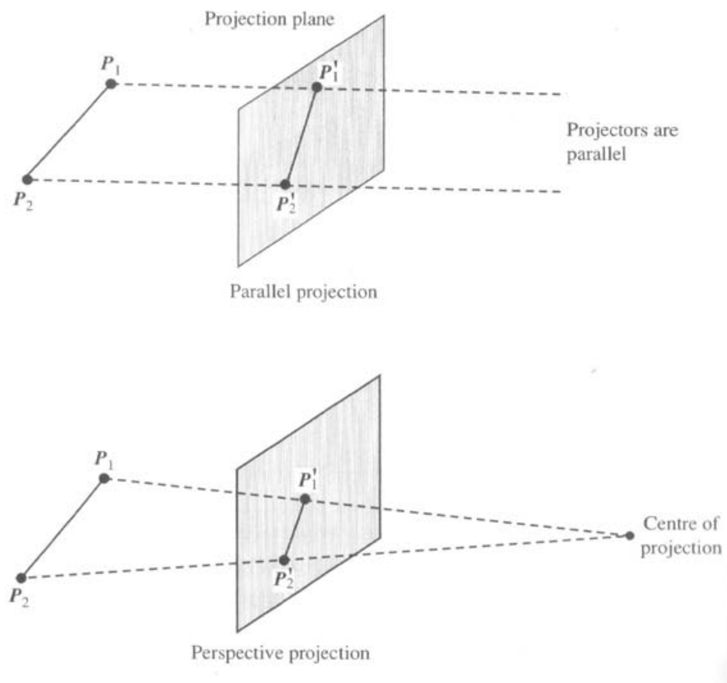


Figure 2-14 The projection in computer graphics [1]

Two points projected onto a plane using parallel and perspective projections.

A perspective projection is the more popular or common choice in computer graphics because it incorporates foreshortening. In a perspective projection relative dimensions are not preserved, and a distant line is displayed smaller than a nearer line of the same length (Figure 2.15). This effect enables human beings to perceive depth in a two-dimensional photograph or a stylization of three-dimensional reality. A perspective projection is characterized by a point known as the centre of projection and the projection of three-dimensional point onto the view plane is the intersection of the line from each point to the centre of projection. These lines are called projectors.

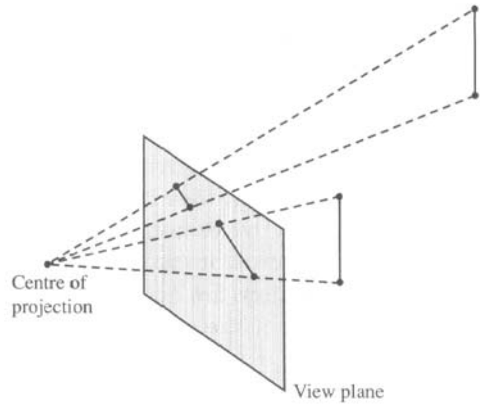


Figure 2-15 The perspective projection [1]

In a perspective projection a distant line is displayed smaller than a nearer line the same length.

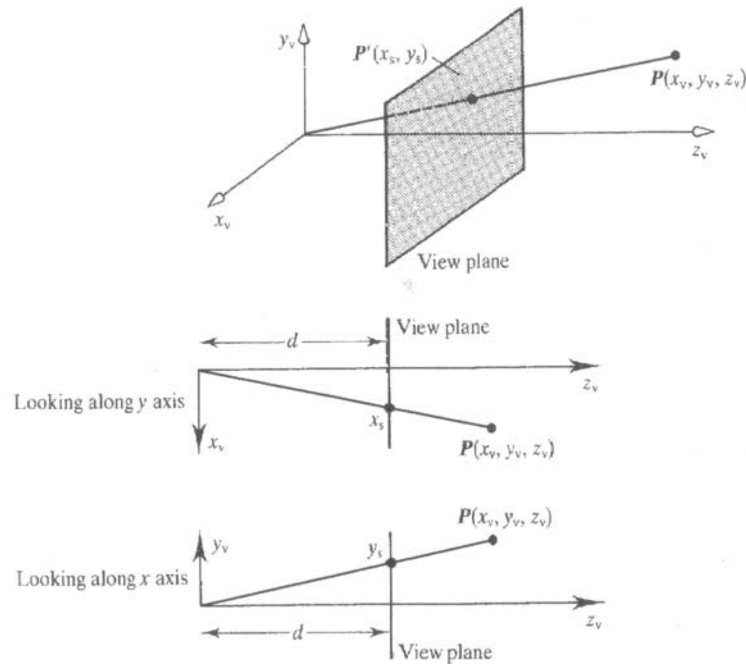


Figure 2-16 Deriving a perspective transformation [1]

Figure 2.16 show how a perspective projection is derived. Point **P** (x_v, y_v, z_v) is a three-dimensional point in the view coordinate system. This point is to be projected onto a view plane normal to the z_v axis and positioned at distant *d* from the origin of this system. Point P' is the projection of this point in the view plane and has two-dimensional coordinates (x_s, y_s) in a view plane coordinate system with the origin at the intersection of the z_v axis and the view plane.

To express this non-linear transformation as a 4x4 matrix we can consider it in two-parts – a linear part followed by a non-linear part. Using homogeneous coordinates we have:

$$
\begin{aligned}
X &= x_v \\
Y &= y_v \\
Z &= z_v \\
w &= z_v / d
\end{aligned}
\tag{19}
$$

We can now write:

$$
\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = T_{pers} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}
\tag{20}
$$

where:

$$
T_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}
\tag{21}
$$

following this with the perspective divide, we have:

$$
\begin{aligned}
x_s &= X / w \\
y_s &= Y / w \\
z_s &= Z / w
\end{aligned}
\tag{22}
$$

In a parallel projection, if the view plane is normal to the direction of projection then the projection is orthographic and we have:

$$
x_s = x_v \quad y_s = y_v \quad z_s = z_v
\tag{23}
$$

Expressed as a matrix:

$$
T_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\tag{24}
$$

2.2.4 Rasterization

Having looked at how general points within a polygon can be assigned intensities

that are determined from vertex values, we now look at how we determine the actual pixels which we require intensity values for. The process is known as rasterization or scan conversion. We consider this somewhat tricky problem in two parts. First, how do we determine the pixels which the edge of a polygon straddles? Second, how do we organize this information to determine the interior points?

Rasterizing edges

There are two difference ways of rasterizing an edge, based on whether line drawing or solid area filling is being used. Line drawing is not covered here, since we are interested in solid object. However, the main feature of line drawing algorithm (for example, Bresenham's algorithm (Bresenham 1965)) is that they must generate a linear sequence of pixels with no gaps (Figure 2.17). For solid area filling, a less rigorous approach suffices. We can fill a polygon using horizontal line segments; these can be thought of as the intersection of the polygon with a particular scan line. Thus, for any given scan line, what is required is the left- and right-hand limits of segment that is the intersections of the scan line with left- and right-hand polygon edges. This means that for each edge's intersections with the scan lines (Figure 2.17 b). This sequence may have gaps, when interpreted as a line, as shown by the right-hand edge in the diagram.

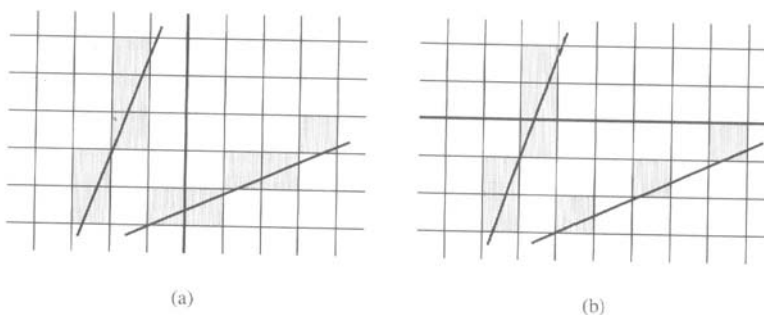


Figure 2-17 The concept of Bresenham's algorithm [1]

Pixel sequences required for (a) line drawing and (b) polygon filling

The conventional way of calculating these pixels coordinates is by use of what is grandly referred to as a 'digital differential analyzer', or DDA for short. All this really consists of is finding how much the x coordinate increases per scan line, and then repeatedly adding this increment.

Let (x_s, y_s), (x_e, y_e) be the start and end points of the edge (we assume that

$y_e > y_s$). The simplest algorithm for rasterizing sufficient for polygon edges is:

$$
\begin{aligned}
& x := x_s \\
& m := (x_e - x_s)/(y_e - y_s) \\
& \text{for} \quad y := y_s \quad \text{to} \quad y_e \quad \text{do} \\
& \qquad output(round(x), y) \\
& \qquad x := x + m
\end{aligned}
$$

The main drawback of this approach is that m and x need to be represented as floating point values, with a floating point addition and real-to-integer version each time round the loop. A method due to Swanson and Thayer (Swanson and Thayer 1986) provides an integer-only version of this algorithm. It can be derived from the above in two logical stages. First we separate out x and m into integer and fractional parts. Then each time round the loop, we separate add two parts, adding a carry to the integer part should the fractional part overflow. Also, we initially set the fractional part of x to -0.5 to make rounding easy, as well as simplifying the overflow condition. In pseudocode:

$$
\begin{aligned}
& xi := x_s \\
& xf := -0.5 \\
& mi := (x_e - x_s) div (y_e - y_s) \\
& mf := (x_e - x_s)/(y_e - y_s) - mi \\
& for \quad y := y_s \quad to \quad y_e \quad do \\
& \qquad output(xi, y) \\
& \qquad xi := xi + mi \\
& \qquad xf := xf + mf \\
& \qquad if \quad xf > 0.0 \quad then \quad \{xi := xi + 1; xf := xf - 1.0\}
\end{aligned}
$$

Because the fractional part is now independent of the integer part, it is possible to scale it throughout by $2(y_e - y_s)$, which the effect of converting everything to integer arithmetic:

$$xi := x_s$$
$$xf := -(y_e - y_s)$$
$$mi := (x_e - x_s) div (y_e - y_s)$$
$$mf := 2*[(x_e - x_s) \bmod (y_e - y_s)]$$
$$for \quad y := y_s \quad to \quad y_e \quad do$$
$$\quad output(xi, y)$$
$$\quad xi := xi + mi$$
$$\quad xf := xf + mf$$
$$\quad if \quad xf > 0 \quad then \quad \{xi := xi + 1; xf := xf - 2(y_e - y_s)\}$$

Although this approach now to involve two divisions rather than one, they are both integer rather than floating point. Also, given suitable hardware, they can both be evaluated from the same division, since the second (mod) is simply the remainder from the first (div). Finally it only remains to point out that the $2(y_e - y_s)$ within the loop is constant and would in practical be evaluated just once outside it.

Rasterizing polygons

Now that we know how to find pixels along the polygon edges, it is necessary to turn our attention to filling the polygons themselves. Since we are concerned with shading, 'filling a polygon' means finding the pixel coordinates of interior points and assigning to these a value calculated using one of the incremental shading schemes described in 2.2.5. We need to generate pairs of segment end points and fill in horizontally between them. This is usually achieved by constructing an 'edge list' for each polygon.

In principle this is done using an array of linked lists, with an element for each scan line. Initially all the elements are set to NIL. Then each edge of the polygon is rasterized in turn, and the *x* coordinate of each pixel (x, y) thus generated is inserted into the linked list corresponding to that value of *y*. Each of the linked lists is then sorted in order of increasing *x*. The result is something like that shown in Figure 2.18. Filling-in of the polygon is then achieved by, for each scan line, taking successive pairs of *x* values and filling in between them (because a polygon has to be closed, there will always be an even number of elements in the linked list). Note that this method is powerful enough to cope with concave polygons with holes.

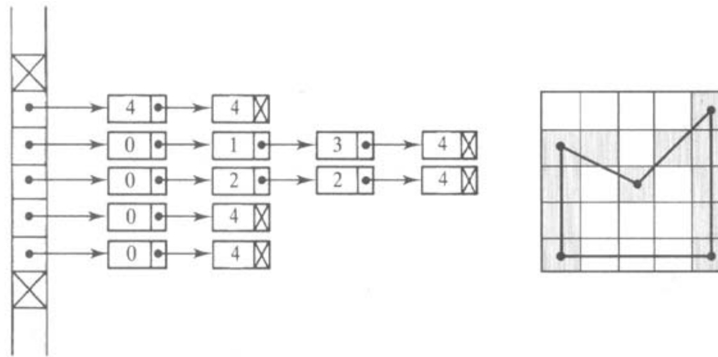


Figure 2-18 The linked list [1]

An example of a linked list maintained in polygon rasterization.

In practice, the sorting of the linked lists is achieved by inserting values in the appropriate place initially, rather than by a big sort at the end. Also, as well as calculating the x value and sorting it for each pixel on an edge, the appropriate shading values would be calculated and stored at the same time (for example, intensity value for Gouraud shading; x, y and z components of the interpolated normal vector for Phong shading).

If the object contains only convex polygons then the linked x lists will only ever contain two x coordinates; the data structure of the edge list is simplified and there is no sort required. It is not a great restriction in practical computer graphics to constrain an object to convex polygons.

One thing that has been slightly glossed over so far is the consideration of exactly where the borders of a polygon lie. This can manifest itself in adjacent polygons either by gaps appearing between them, or by them overlapping. For example, in Figure 2.19, the width of the polygon is 3 units, so it should have an area of 9 units, whereas it has been rendered with an area of 16 units. The traditional solution to this problem, and the one usually advocated in textbook, is to consider the sample point of the pixel to lie in its centre, that is, at $(x+0.5, y+0.5)$. (A pixel can be considered to be a rectangle of finite area with dimensions 1.0*1.0, and its sample point is the point within the pixel area where the scene is sampled in order to determine the value of the pixel.) So, for example, the intersection of an edge with a scan line is calculated for y+0.5, rather than for y, as we assumed above. This is messy, and excludes the possibility of using integer-only arithmetic. A simpler solution is to assume that the sample point lies

at one of the four corners of the pixel; we have chosen the top right-hand corner of the pixel. This has the consequence that the entire image is displaced half a pixel to the left and down, which in practice is insignificant. The upshot of this is that it provides the following simple Rasterization rules:

(1) Horizontal edges are simply discarded.

(2) An edge which goes from scan line y_{bottom} to y_{top} should generated x values for scan lines y_{bottom} through to $y_{top}-1$ (that is missing the top scan line), or if $y_{bottom}=y_{top}$ then it generates no values.

(3) Similarly, horizontal segments should be filled from x_{left} to $x_{right}-1$ (with no pixels generated if $x_{left}=x_{right}$).

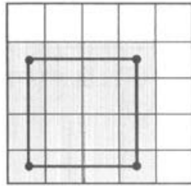


Figure 2-19 The problem with polygon boundaries [1]

Problems with polygon boundaries – a 9-pixel polygon fills 16 pixels.

Incidentally, in rule (2) and (3), whether the first or last element is ignored is arbitrary, and the choice is based around programming convenience. The four possible permutations of these two rules define the sample point as one of the four corners of the pixel. The effect of these rules can be demonstrated in Figure 2.20. Here we have three adjacent polygons A, B and C, with edges a, b, c, and d. the rounded x values produced by these edges for the scan shown are 2, 4, 4, and 7 respectively. Rule 3 then gives pixels 2 and 3 for polygon A, none for polygon B, and 4 to 6 for polygon C. Thus, overall, there are no gaps, and no overlapping. The reason why horizontal edges are discarded is because the edges adjacent to them will have already contributed the x values to make up the segment (for example, the base of the polygon in Figure 2.18; note also that, for the sake of simplicity, the scan conversion of this polygon was not done strictly in accordance with the Rasterization rules mentioned above).

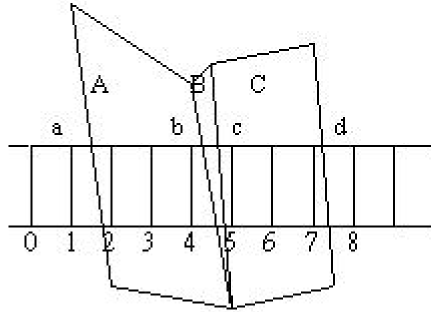


Figure 2-20 The result of Rasterization rules [1]

Three polygons intersecting a scan line.

2.2.5 Shading

Interpolative shading techniques

Having dealt with the problem of calculating light intensity at a point, we now consider how to apply such a model to a polygon and calculate the light intensity over its surface. Two classic techniques have emerged – Gouraud and Phong shading. Phong interpolation gives the more accurate highlights – as we shall see – and is generally the preferred model. Gouraud shading on the other hand is considerably cheaper. Both techniques have been developed both to interpolate information efficiently across the face of a polygon and to diminish the visibility of the polygon edges in the final shading image. Information is interpolated from values at the vertices of a polygon and the situation is exactly analogous to depth interpolation.

Interpolative shading techniques – Gouraud shading

In Gouraud shading we calculate light intensity – using the local reflection model – at the vertices of the polygon and then interpolation between these intensities to find values at projected pixels. To do this we use the bilinear interpolation equations, the property *p* being the vertex intensity I. The particular surface normals used at a vertex are special normals called vertex normals. If we consider a polygon in isolation then, of course, the vertex normals are parallel. However, in Gouraud shading we use special normals called vertex normals and it is this device that reduces the visibility of polygon edges. Consider Figure 2.21. Here the vertex normal $\overline{N_A}$ is calculated by averaging $\overline{N_1}$, $\overline{N_2}$, $\overline{N_3}$ and $\overline{N_4}$.

$$\overline{N_A} = \overline{N_1} + \overline{N_2} + \overline{N_3} + \overline{N_4} \tag{25}$$

$\overline{N_A}$ is then used to calculated an intensity at vertex A that is common to all the polygons that share vertex A.

For computational efficiency the interpolation equations are implemented as incremental calculations. This is particularly important in the case of the third equation, which is evaluated for every pixel. If we define Δx to be the incremental distance along a scan line then ΔI, the change in intensity from one pixel to the next, is:

$$\Delta I = \frac{\Delta x}{x_b - x_a}(I_b - I_a)$$
$$I_{s,n} = I_{s,n-1} + \Delta I_s \tag{26}$$

Because the intensity is only calculated at vertices the method cannot adequately deal with highlights and this is its major disadvantage. The cause of this defect can be understood by examining Figure 2.22a. We have to bear in mind that the polygon mesh is an approximation to a curved surface. For a particular viewing and light source direction we can have a diffuse component at A and B and a specular highlight confined to some region between them. Clearly if we are deriving the intensity at pixel P from information at A and B we will not calculate a highlight. This situation is nearly taken care of by interpolating vertex normals rather than intensities as shown in Figure 2.22b. This approach is know as Phong shading.

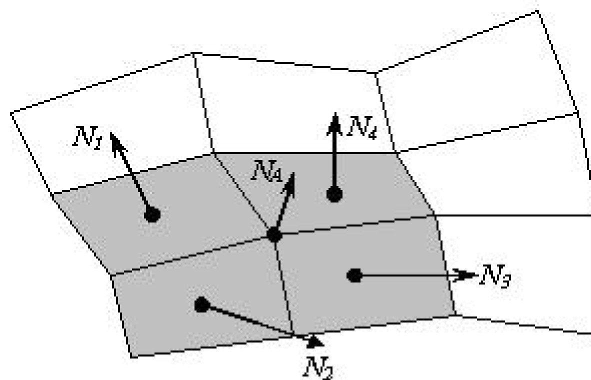


Figure 2-21 The Gouraud Shading [1]

The vertex normal $\mathbf{N}_A$ is the average of the normals $\mathbf{N}_1$, $\mathbf{N}_2$, $\mathbf{N}_3$ and $\mathbf{N}_4$, the normals of the polygon that meet at the vertex.

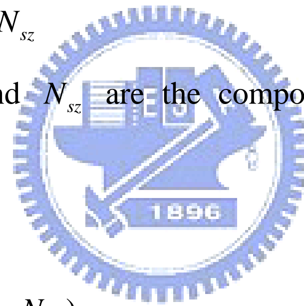
Interpolative shading techniques – Phong shading

Here we interpolate vertex normals across the polygon interior and calculate for each polygon pixel projection an interpolated normal. This interpolated normal is then used in the shading equation which is applied for every pixel projection. This has the geometric effect (Figure 2.22) of 'restoring' some curvature to polygonally faceted surface.

The price that we pay for this improved model is efficiency. Not only is the vector interpolation three times the cost of intensity interpolation, but each vector has to be normalized and a shading equation calculated for each pixel projection.

Incremental computation can be employed as with intensity interpolation, and the interpolation would be implemented as:

$$\begin{aligned} N_{sx,n} &= N_{sx,n-1} + \Delta N_{sx} \\ N_{sy,n} &= N_{sy,n-1} + \Delta N_{sy} \\ N_{sz,n} &= N_{sz,n-1} + \Delta N_{sz} \end{aligned} \tag{27}$$



Where N_{sx}, N_{sy} and N_{sz} are the components of a general scan line normal vector N_s and:

$$\Delta N_{sx} = \frac{\Delta x}{x_b - x_a}(N_{bx} - N_{ax}) \tag{28}$$

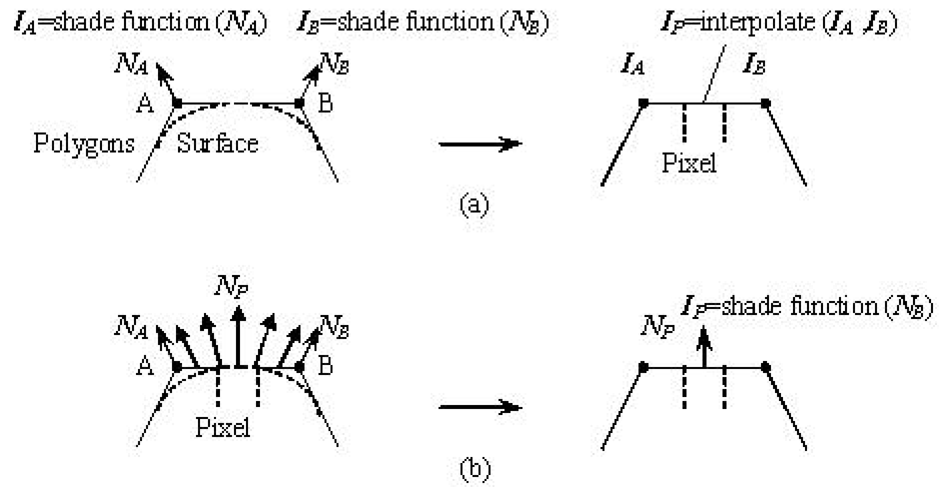


Figure 2-22 The difference between Gouraud and Phong shading [1]

Illustrating the difference between Gouraud and Phong shading.

(a) Gouraud shading

(b) Phong shading

Chapter 3 Design

In this chapter, we analyze the operations, parallelism and computation flow for rendering process and propose a run-time reconfiguration scheduling.

3.1 Analysis

Now we know how the rendering process goes. It is shown in Figure 3.1. Data flow graph of each stage would be shown in Figure 3.2.

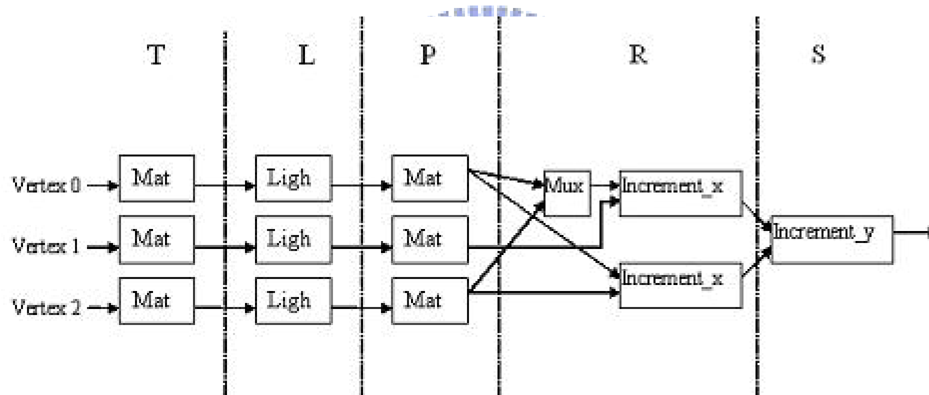


Figure 3-1 The example of Rendering process

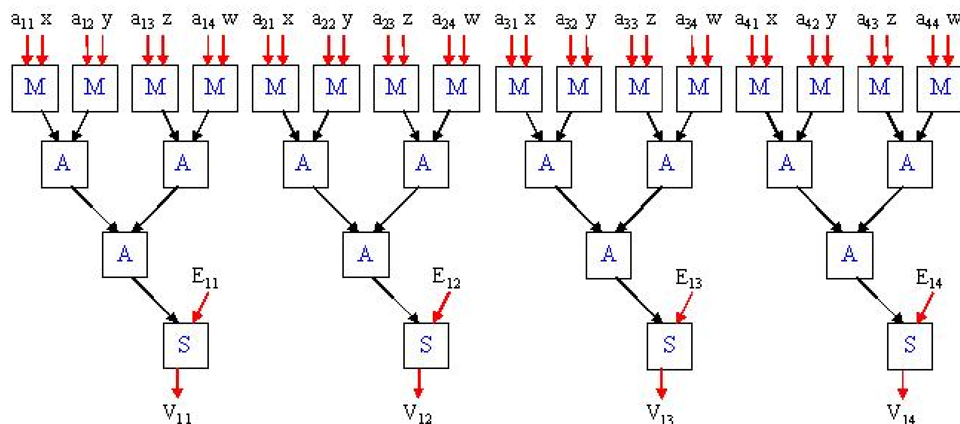
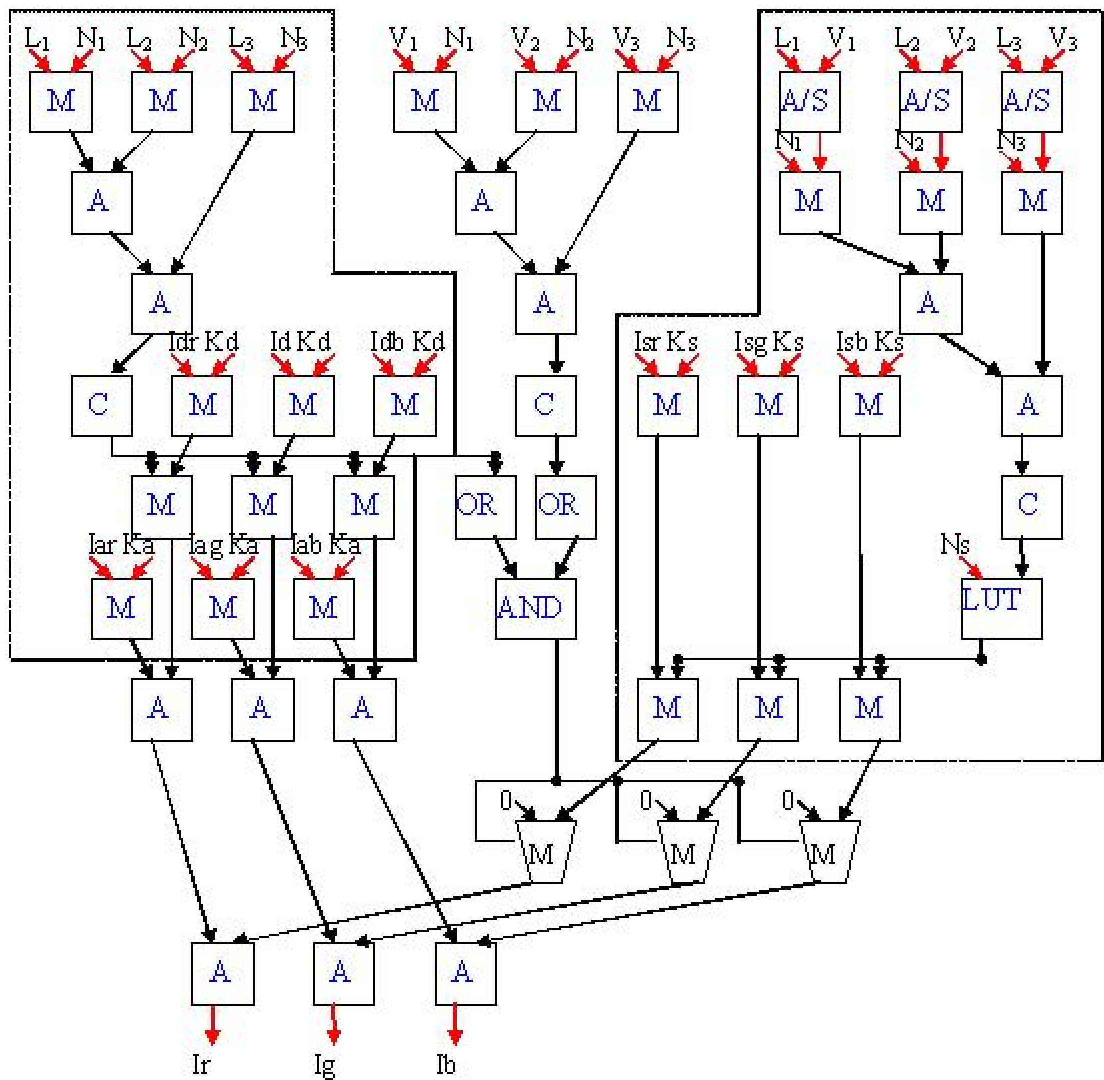
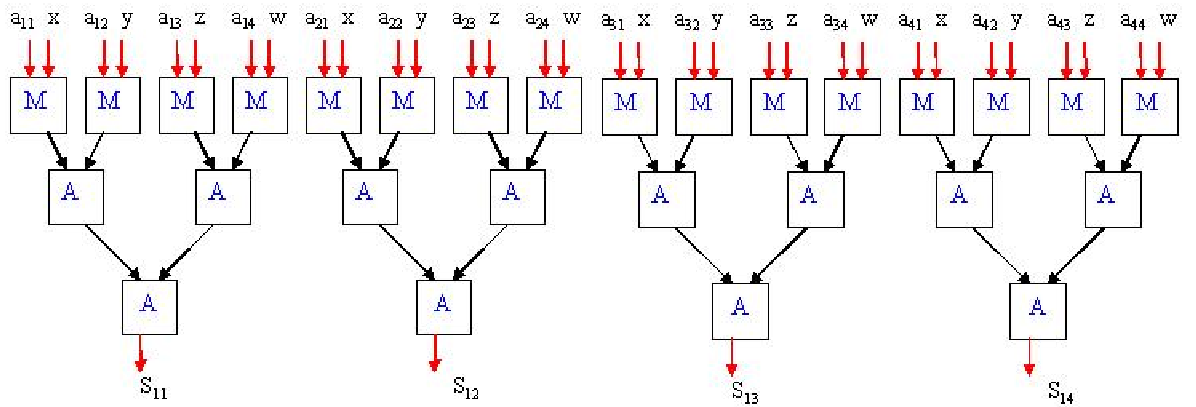


Figure 3-2 The DFG of one vertex

(a) Transformation DFG of one vertex



(b) Lighting DFG of one vertex



(c) Projection DFG of one vertex

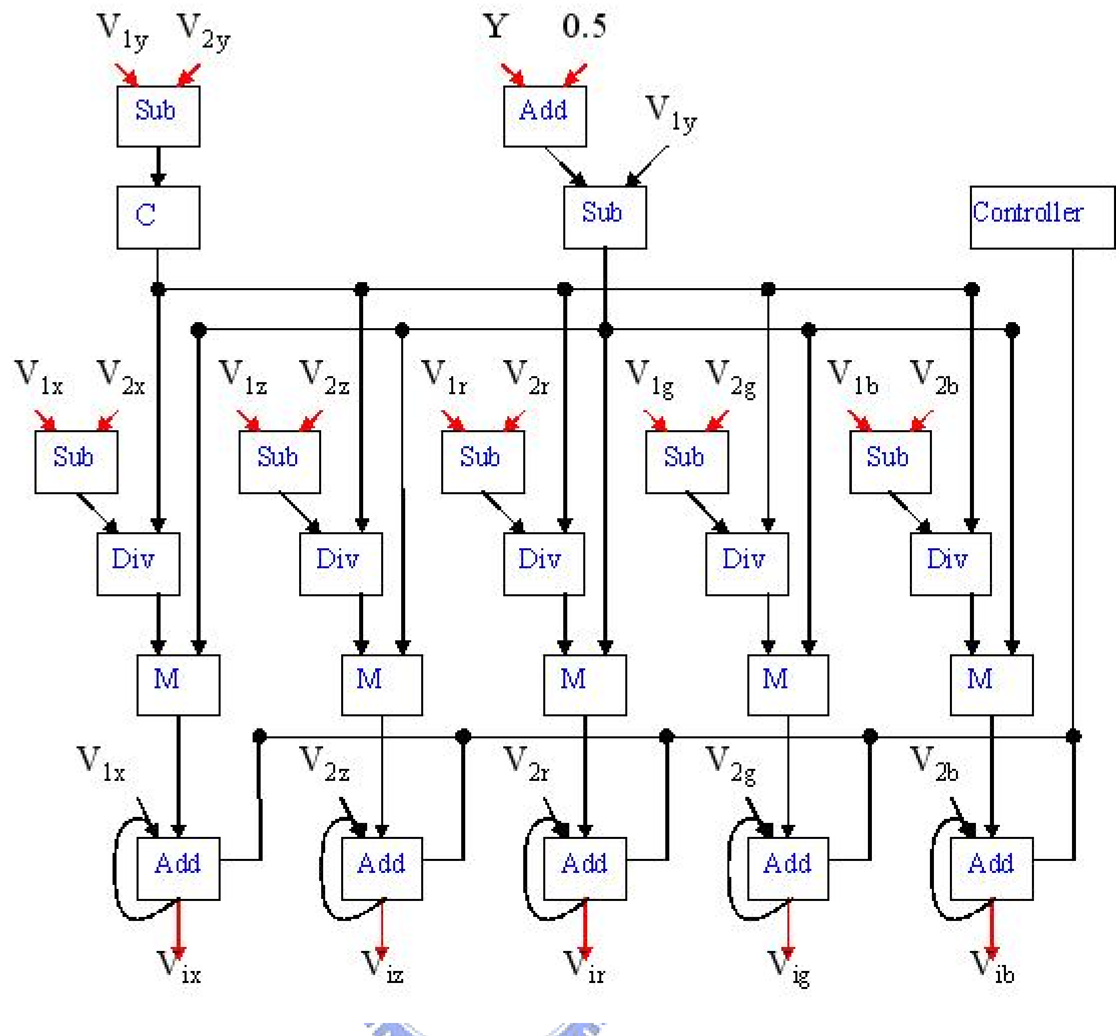
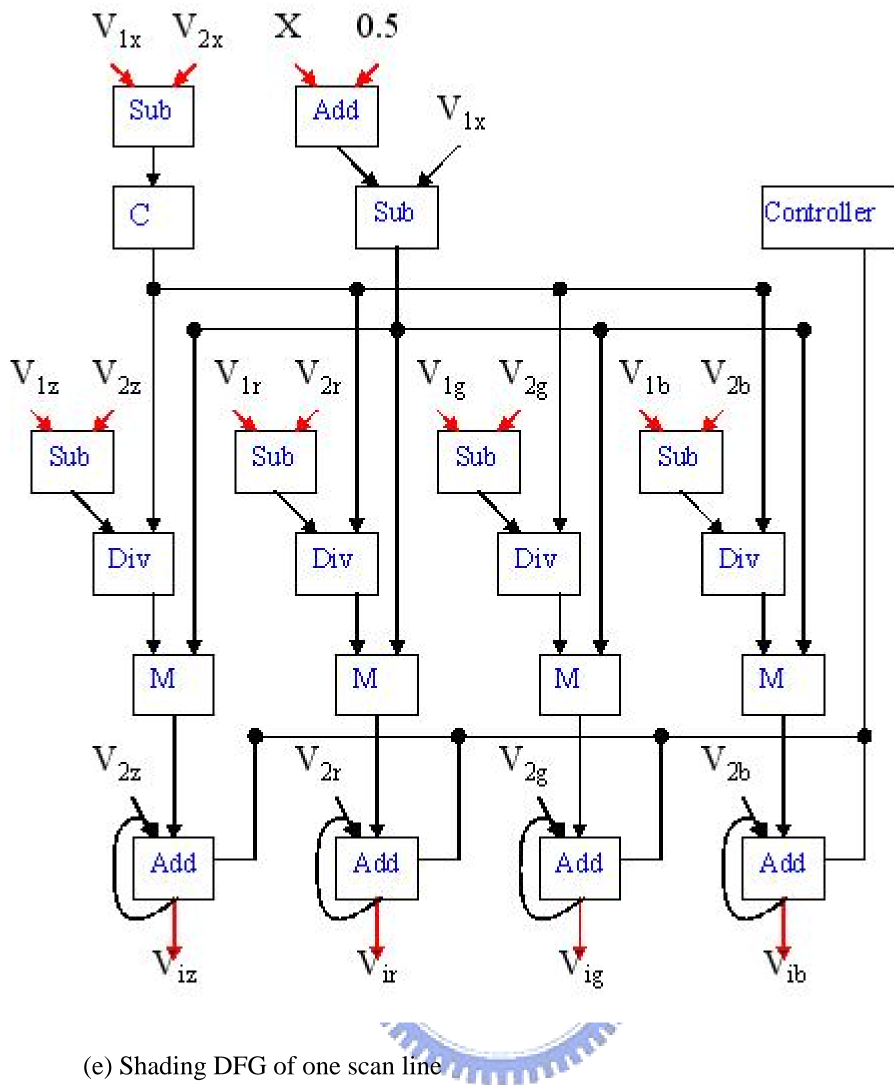


Figure3.2
(e) Rasterization DFG of one line



(e) Shading DFG of one scan line

Basic requirement of arithmetic units is listed in the Table 3.1.

Table 3-1 Basic operation for each stage

	Mul	Add	sub	div	other	total	Process unit
Transformation	4*4	3*4	1*4	0	0	32	Vertex
Lighting	24	15	0	0	10	49	Vertex
Projection	4*4	3*4	0	0	0	28	Vertex
Rasterization	5	6	7	5	2	25	Edge
Shading	4	5	6	4	2	21	Pixel

In the Table 3.1, we can see that we need the operation of multiplication, division, add, subtract and other (shift, AND, OR, compare, LUT…). 3D computer graphics

applications are usually 8-bits, 12-bits and above operations, so I think the coarse-grained architecture would be suitable for my design. As we can see, we need multiplication and division operations as much as add and subtract operations, hence I assume the Processing Element (PE) has ability to process multiplication, division, add and subtract with multiplication and division operations take multiple cycles and add and subtract operations are single cycle execution. Besides PE also can do other operations (shift, AND, OR, compare, LUT…). Hence execution time and resource of each stage with one polygon is shown in Table 3.2.

Table 3-2 Resource and execution time of each stage

	PEs	Execution Time
Transformation	32*3=96	M+3
Lighting	49*3=147	3M+3
Projection	28*3=84	M+2
Rasterization& Shading	25*2+21=71	2M+2+Rn*(2M+2+Sn)

In Table 3.2, M means multiplication execution cycles of one PE and Rn, means pixels generated from Rasterization stage and Sn means pixels generated from shading stage.

3.2 Design

For now we know that execution time and resource of processing one polygon, we can consider how to process multiply polygons.

Fixed Configuration Design :

It is common to think that we give a fixed configuration for each stage and the execution model would be shown in Figure 3.3. It is a pipeline design with unbalance stages. Here I do not focus on finding a balance pipeline design for the rendering application and one nature of Rasterization/Shading stage will

cause the pipeline design unbalance.

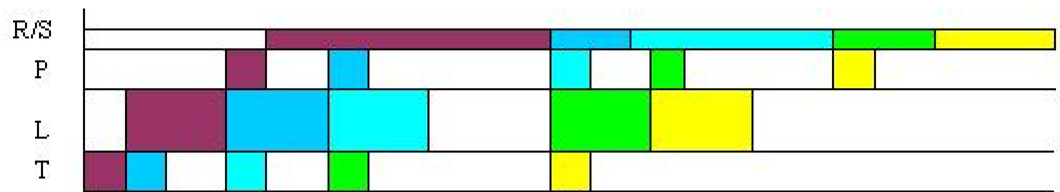


Figure 3-3 The execution example of fixed configuration design

3.3 Observation

In 3D computer graphics application, execution cycles of Rasterization/Shading stage is not constant because a polygon size projected to the view plane would not be the same with other polygons. For this characteristic, if there is no buffer between each two stage, it will cause some stage to wait using fixed configuration design. Besides, total execution time is usually dominated by the R/S stage.

3.4 Opportunity of improvement

According to the observation, we can improve the fixed configuration design by adding buffers between each two stages to solve stage waiting problem and we can use multiple copies of stage hardware if one stage executed time is long. Hence the execution model of modified fixed configuration design would be shown in Figure 3.4.

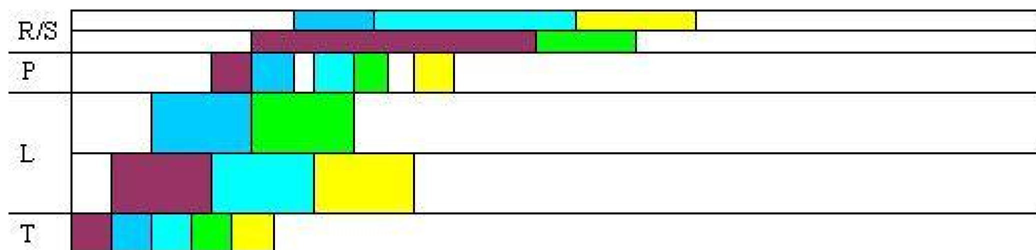


Figure 3-4 The execution example of modified fixed configuration design

In the other hand, we can use Runt-time Reconfiguration design to improve original fixed configuration design. Run-time Reconfiguration is based upon the concept of virtual hardware, which is similar to virtual memory. In fixed configuration design, we

have fixed configuration of each stage unless we redesign it but in Run-time Reconfiguration design we only need to reconfigure the reconfigurable hardware into the stage we want. The disadvantage is that reconfigurable hardware resource is limited. If we use more hardware resource for one stage, it means less hardware resource will be left for other stages. Hence scheduling is important for Run-time Reconfiguration design. The execution model example would be shown in Figure 3.5.

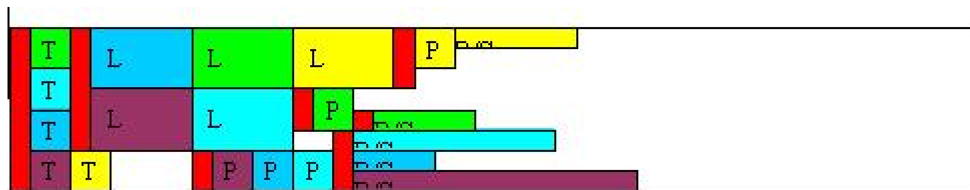


Figure 3-5 The execution example of Run-time Reconfiguration design

3.5 Dynamic Task Scheduling

A dynamic task scheduling for Run-time Reconfiguration design is executed by a controller in Reconfigurable system.

In the rendering applications, one polygon would be processed through four stages (Transformation, Lighting, Projection and Rasterization/Shading) and every polygon is ready for T stage at the beginning and can be parallel processed. Because there is a reconfiguration overhead using Run-time Reconfiguration design and buffer size is limited, so the scheduling should satisfy:

1. If buffer size is full, reconfigure the hardware to process those polygon in buffer
2. Minimize configuration overhead

Here I propose a best fit scheduling which process polygons of buffer size. It will execute like this. First, partition all polygons into several sets which are as the same size as buffer size. Second, we configure the entire reconfigurable hardware into several transformation configurations and parallel process one set polygons as much as hardware can. Third, we configure the entire reconfigurable hardware into several lighting stage configurations and parallel process those polygons which are generated by previous stage in the buffer. Then process those polygons in projection stage and next rasterization/shading stage using similar scheduling. When one set of polygons finishes, next set of polygons will be processed in the same way until all polygon finish.

Chapter 4 Simulation

In this Chapter, we will describe the simulation environment and show the simulation result of total execution time under several PE number.

4.1 Simulation Environment

In this thesis, we assume our PE has ability to do add, subtraction, multiplier and division with add and subtraction both are single cycle execution but multiplier and division are multiple cycle execution. Here we assume multiplier and division take five cycles to finish the operation because each PE is an 8-bit ALU. Another assumption is that each configuration takes 1000 cycle. Nowadays average configuration time is ranged form nanosecond to microsecond. If our reconfigurable hardware runs at 100MHz, 1000 cycles are equal to 10 microseconds. It is conservative assumption. Finally, we use multicontext reconfigurable system with partially reconfigurable models and place & route problem is ignored in this thesis.

4.2 Simulation Result

We choose Venus, UTAH teapot and Bunny69k model for benchmark. They are very familiar in computer graphics. I will simulate different PE number and different buffer size to see the influence to the reconfigurable system. The parameters of each benchmark are listed in Table 4.1. Scales means that the size projected to the view plane of the benchmark. Different scales of same benchmark have different execution time of rasterization/shading stage and we are going to see what is will cause in our simulation.

Table 4-1 Benchmarks

	Vertices	Scale
Venus	1396	1
Utah Teapot	6400	5
	6400	10
	6400	20
Bunny69k	69451	2

In original fixed configuration design, it use 398 PEs(96 for transformation, 147 for lighting, 84 for projection and 71 for rasterization/shading) for execution, each stage is executed one polygon at a time. In the next, I simulate modified fixed configuration design for different PE numbers and the result is listed in Table 4.2.

Table 4-2 Execution time comparison 1

	Venus	teapot-5	teapot_10	teapot-20	bunny69k
(1,1,1,1)	755402	134436	170544	442977	1997474
(1,1,1,2)	377845	134436	134439	227994	1458507
(1,1,1,3)	251979		134439	157645	1458507
(1,1,1,4)	189154			134465	
(1,1,1,5)	151232			134465	
(1,1,1,10)	76314				
(1,1,1,20)	40339				
(1,1,1,30)	32574				
(1,1,1,31)	32451				
(1,1,1,32)	32451				

First column represent number of fixed hardware (transformation, lighting, projection, rasterization/shading). Two to four copies of rasterization/shading hardware are enough but it needs thirty-one copies in Venus model that is because projected size of every polygon in Venus model on view plane is much bigger than other models. After this simulation, I found another bottleneck that lies in light stage. Hence I use two copies lighting hardware to reduce average execution cycle of a polygon in lighting. Because execution cycles of a polygon in transformation stage and projection stage are 8 and 7 cycles but execution cycles of one polygon in lighting stage is 21 cycles. After I use two copies of lighting stage hardware, execution cycle of a polygon in lighting stage will reduce to 10.5 cycle approximate to 8 or 7 cycles than 21 cycles. It makes data flow

more smoothly but it cost more hardware. The result is listed in Table 4.3.

Table 4-3 execution time comparison 2

	venus	teapot-5	teapot_10	teapot-20	bunny69k
(1,2,1,1)	755402	106866	166607	442977	1997075
(1,2,1,2)	377838	67240	85294	221511	998771
(1,2,1,3)	251968	67240	67247	149247	729284
(1,2,1,4)	189137		64247	114037	729284
(1,2,1,5)	151410			92921	
(1,2,1,6)	126211			78860	
(1,2,1,7)	108451			68816	
(1,2,1,8)	94995			67273	
(1,2,1,9)	84573			67273	
(1,2,1,10)	76265			67273	
(1,1,1,20)	39840				
(1,1,1,30)	27908				
(1,1,1,40)	22000				
(1,1,1,50)	19375				
(1,1,1,55)	18556				
(1,1,1,57)	18338				
(1,1,1,59)	18199				
(1,1,1,60)	18150				
(1,1,1,61)	18150				

We can see total execution time decrease by adding one copies of lighting hardware. Once if we want to improve performance, we can just add more hardware but cost may goes to high. It means that we add double hardware but we may not decrease half execution time. We can use the cost function $\alpha(Time) \times \beta(hardware)$ to represent this and hardware here means PE number. Time in fixed configuration design only means execution time but it represents execution time and reconfiguration time in Run-time Reconfiguration design. Hence the cost function will become $\alpha(T_{exe} + T_{rec}) \times \beta(hardware)$.

In Run-time Reconfiguration design, I simulate every ten PE numbers from 147 PE numbers to 497 PE numbers. Because bunny69k model execution cycles are much more than others, I use an independent figure to show avoiding other result can't be seen clearly. Figure 4.1a and 4.1b are the execution cycles of each models.

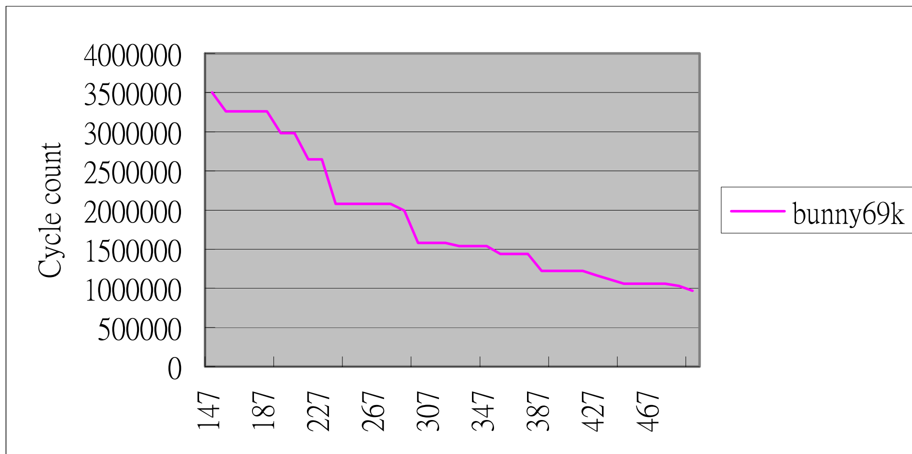
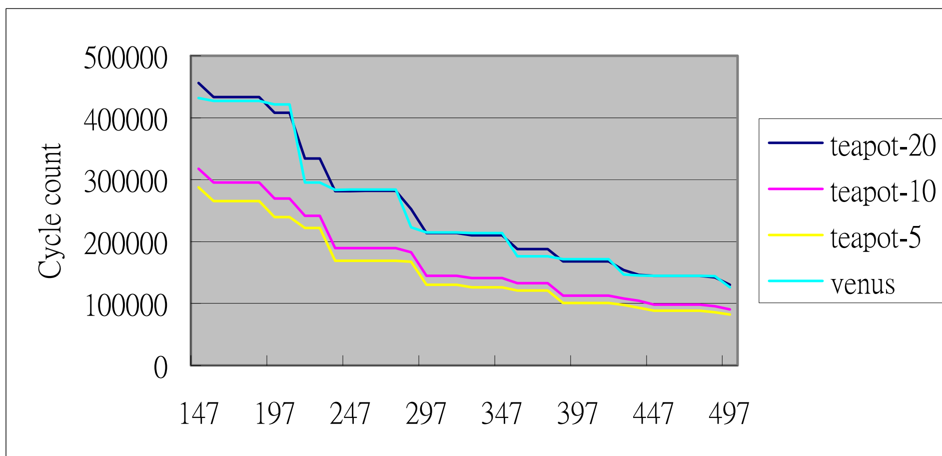


Figure 4-1 The simulation result of different PE

(a) Cycle count of different PE for bunny69k



(b) Cycle count of different PE for other

Figure 4.2 shows the cost function:

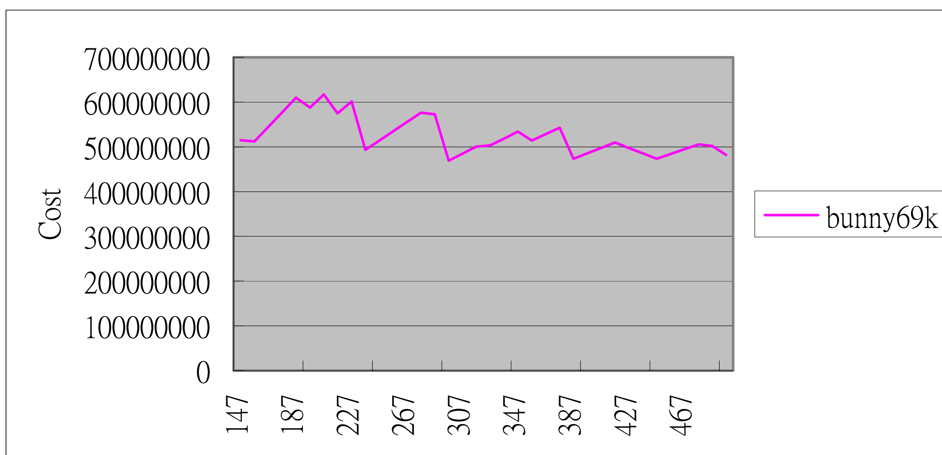
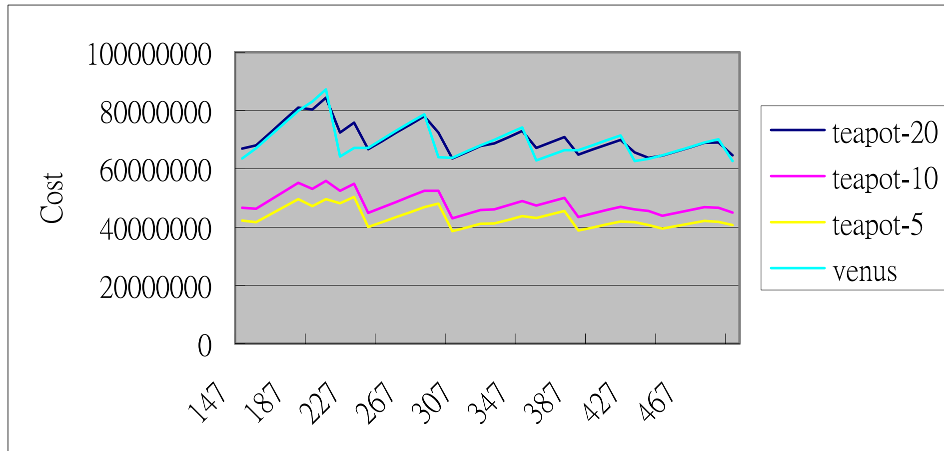


Figure 4-2 The simulation result of cost function

(a) Cost function of different PE for bunny69k



(b) Cost function of different PE for other

Lowest cost of each model are at PE numbers is 297 except Venus model is 427 that is the same reason as more copies of rasterization/shading stage hardware. I compare lowest cost of Run-time Reconfiguration and modified fixed configuration design. The result is listed in Table 4.4.

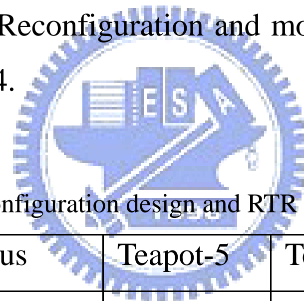


Table 4-4 The comparison of fixed configuration design and RTR

		Venus	Teapot-5	Teapot-10	Teapot-20	Bunny69k
Modified fixed configuration design	PE	2604	616	687	971	687
	Cycle	27908	67240	64247	67273	729284
	cost	72672432	41419840	46198689	66820336	501018108
Run-time reconfiguration design	PE	427	297	297	297	297
	Cycle	146695	129921	144854	213951	1579774
	cost	62638765	38586537	43021638	63543447	469192878

Chapter 5 Conclusion and Future work

In this chapter, according to previous discussions, we have some conclusions and find some future works for research.

5.1 Conclusion

The hardware accelerated architectures for computer graphics now have two approaches. One is fixed configuration design another is Run-time Reconfiguration design. From the cost function, Run-time Reconfiguration design wins a little bit. If you want to find a very short execution time solution, there is no doubt that fixed configuration design is what you looking for. But if you want a low cost solution, Run-time Reconfiguration design may be suitable for your needs. There is another advantage for Run-time Reconfiguration design, flexibility. In this thesis, flexibility means we can reconfigure the reconfigurable hardware between 4 stages (transformation, lighting, projection and rasterization/shading). That is why we can use smaller hardware to execute massive computations. We even can use it to accelerate other algorithms only if we have their configurations which are complied configurations or manually generated.

5.2 Future work

First, place and route problem we ignored is a critical issue if we want to implement a reconfigurable system. During execution, run-time relocation of PE and data transfer can not be ignored. Second, configuration time problem is another issue to the performance. If we can reduce configuration time by using configuration prefetching

and configuration compression, total execution time will be reduced. Last, if configurations can be generated automatically by the compiler, we can accelerate more than rendering process and previous two points we talk about are the work of the compiler. The issue of every reconfigurable system is the lack of a good compiler. Hence the compiler of reconfigurable system is a good topic of research.

Reference

[1]: Alan Watt, 3D Computer Graphics, Third edition, Addison-Wesley, USA, 2000.
[2]: Katherine. Compton, cott. Hauck, "Reconfigurable Computing: A survey of System and software," ACM Computing Survey, June 2002.
[3]: Purna, K. M. G. and Bhatia, D.," Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers", IEEE Trans. Computer, Vol. 48, NO. 6, 579-590, 1999
[4]: Henry Styles, Wayne Luk," *Customising Graphics Applications: Techniques and Programming Interface*", IEEE Symposium on Field-Programmable Custom Computing Machines, 2000
[5]: Arunachalam Ramanathan, Nirupama Ramaswamy, Jeevan Chittamuru, Krishna Prasad Valluru," *Low Power Reconfigurable Core For 3D Graphics Shading and Texture Mapping*", Umass project, 2001
[6]: Pavel Zemick, "Hardware Accleration of Graphics and Imaging Algorithm Using FPGAs", ACM Spring Conference of Computer Graphics, 2002
[7]: EBELING, C., CRONQUIST, D. C., AND FRANKLIN, P. RaPiD—Reconfigurable pipelined datapath. Lecture Notes in Computer Science 1142—"*Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*." R. W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, 126–135, 1996.
[8]: CADAMBI, S., WEENER, J., GOLDSTEIN, S. C., SCHMIT, H., AND THOMAS, D. E. "*Managing pipeline reconfigurable FPGA,*" ACM/SIGDA International Symposium on FPGAs, 55–64, 1998.
[9]: RUPP, C. R., LANDGUTH, M., GARVERICK, T., GOMERSALL, E., HOLT, H., ARNOLD, J. M., AND GOKHALE, M. ,"*The NAPA adaptive processing architecture,*" IEEE Symposium on Field-Programmable Custom Computing Machines, 28–37, 1998.
[10]: SCALERA, S. M. AND VAZQUEZ, J. R. "The design and implementation of a context switching FPGA," IEEE Symposium on Field- Programmable Custom Computing Machines, 78–85, 1998.
[11]: HAUCK, S. AND BORRIELLO, G. "Pin assignment for multi-FPGA systems," IEEE Trans. Comput. Aid. Desi. Integ. Circ. Syst. 16, 9, 956–964, 1997.

[12]: XILINX, INC. "*The Programmable Logic Data Book*," Xilinx, Inc., San Jose, CA, 1994.

[13]: ALTERA CORPORATION. "*Data Book*," Altera Corporation, San Jose, CA, 1998.

[14]: LUCENT TECHNOLOGIES, INC. "*FPGA Data Book*," Lucent Technologies, Inc., Allentown, PA, 1998.

[15]: DEHON, A. *"DPGA Utilization and Application,"* ACM/SIGDA International Symposium on FPGAs, 115–121, 1996.

[16]: TRIMBERGER, S., CARBERRY, D., JOHNSON, A., AND WONG, J. *"A time-multiplexed FPGA,"* IEEE Symposium on Field-Programmable Custom Computing Machines, 22–28, 1997.

[17]: CHAMELEON SYSTEMS, INC. "*CS2000 Advance Product Specification,"* Chameleon Systems, Inc., San Jose, CA, 2000.

[18]: HAUCK, S., FRY, T. W., HOSLER, M. M., AND KAO, J. P. *"The Chimaera reconfigurable functional unit,"* IEEE Symposium on Field-Programmable Custom Computing Machines, 87–96, 1997.

[19]: GOLDSTEIN, S. C., SCHMIT, H., BUDIU, M., CADAMBI, S., MOE, M., AND TAYLOR, R. *"PipeRench: A Reconfigurable Architecture and Compiler,"* IEEE Computer, vol. 33, No. 4, 2000.

[20]: XILINX, INC. "*XC6200: Advance Product Specification*," Xilinx, Inc., San Jose, CA, 1996.

[21]: XILINX, INC. "*VirtexTM 2.5 V Field Programmable Gate Arrays: Advance Product Specification*," Xilinx, Inc., San Jose, CA, 1999.