Chapter 2

ALGORITHMICS FOR IMAGE GENERATION

Before going into the details of various image synthesis algorithms, it is worth considering their general aspects, and establishing a basis for their comparison in terms of efficiency, ease of realization, image quality etc., because it is not possible to understand the specific steps, and evaluate the merits or drawbacks of different approaches without keeping in mind the general objectives. This chapter is devoted to the examination of algorithms in general, what has been called algorithmics after the excellent book of D. Harel [Har87].

Recall that a complete image generation consists of model decomposition, geometric manipulation, scan conversion and pixel manipulation algorithms. The resulting picture is their collective product, each of them is responsible for the image quality and for the efficiency of the generation.

Graphics algorithms can be compared, or evaluated by considering the reality of the generated image, that is how well it provides the illusion of photos of the real world. Although this criterion seems rather subjective, we can accept that the more accurately the model used approximates the laws of nature and the human perception, the more realistic the image which can be expected. The applied laws of nature fall into the category of geometry and optics. Geometrical accuracy regards how well the algorithm sustains the original geometry of the model, as, for example, the image of a sphere is expected to be worse if it is approximated by a polygon mesh during synthesis than if it were treated as a mathematical object defined by the

equation of the sphere. Physical, or optical accuracy, on the other hand, is based on the degree of approximations of the laws of geometric and physical optics. The quality of the image will be poorer, for example, if the reflection of the light of indirect lightsources is ignored, than if the laws of reflection and refraction of geometrical optics were correctly built into the algorithm.

Image synthesis algorithms are also expected to be fast and efficient and to fit into the memory constraints. In real-time animation the time allowed to generate a complete image is less than 100 msec to provide the illusion of continuous motion. In **interactive systems** this requirement is not much less severe if the system has to allow the user to control the camera position by an interactive device. At the other extreme end, high quality pictures may require hours or even days on high-performance computers, thus only 20–30 % decrease of computational time would save a great amount of cost and time for the user. To describe the time and storage requirements of an algorithm independently of the computer platform, **complexity measures** have been proposed by a relatively new field of science, called theory of computation. Complexity measures express the rate of the increase of the required time and space as the size of the problem grows by providing upper and lower limits or asymptotic behavior. The problem size is usually characterized by the number of the most important data elements involved in the description and in the solution of the problem.

Complexity measures are good at estimating the applicability of an algorithm as the size of the problem becomes really big, but they cannot provide characteristic measures for a small or medium size problem, because they lack the information of the time unit of the computations. An algorithm having $\text{const}_1 \cdot n^2$ computational time requirement in terms of problem size n, denoted usually by $\Theta(n^2)$, can be better than an algorithm of $\text{const}_2 \cdot n$, or $\Theta(n)$, if $\text{const}_1 << \text{const}_2$ and n is small. Consequently, the time required for the computation of a "unit size problem" is also critical especially when the total time is limited and the allowed size of the problem domain is determined from the overall time requirement. The unit calculation time can be reduced by the application of more powerful computers. The power of general purpose processors, however, cannot meet the requirements of constantly increasing expectations of the graphics community. A real-time animation system, for example, has to generate at least 15 images per second to provide the illusion of continuous motion. Suppose that the number of pixels on the screen is about 10^6 (advanced systems usually have

1280×1024 resolution). The maximum average time taken to manipulate a single pixel (t_{pixel}), which might include visibility and shading calculations, cannot exceed the following limit:

$$t_{\text{pixel}} < \frac{1}{15 \cdot 10^6} \approx 66 \text{ nsec.} \tag{2.1}$$

Since this value is less than a single commercial memory read or write cycle, processors which execute programs by reading the instructions and the data from memories are far too slow for this task, thus special solutions are needed, including:

1. **Parallelization** meaning the application of many computing units running parallelly, and allocating the computational burden between the parallel processors. Parallelization can be carried out on the level of processors, resulting in multiprocessor systems, or inside the processor, which leads to special graphics chips capable of computing several pixels parallelly and handling tasks such as instruction fetch, execution and data transfer simultaneously.

2. **Hardware realization** meaning the design of a special digital network instead of the application of general purpose processors with information about the algorithm contained by the architecture of the hardware, not by a separate software component as in general purpose systems.

The study of the hardware implementation of algorithms is important not only for hardware engineers but for everyone involved in computer graphics, since the requirements of an effective software realization are quite similar to those indispensable for hardware translation. It means that a transformed algorithm ready for hardware realization can run faster on a general purpose computer than a naive implementation of the mathematical formulae.

2.1 Complexity of algorithms

Two complexity measures are commonly used to evaluate the effectiveness of algorithms: the **time** it spends on solving the problem (calculating the result) and the size of **storage** (memory) it uses to store its own temporary

data in order to accelerate the calculations. Of course, both the time and storage spent on solving a given problem depend on the one hand on the *nature of the problem* and on the other hand on the *amount* of the input data. If, for example, the problem is to find the greatest number in a list of numbers, then the size of the input data is obviously the length of the list, say n. In this case, the time complexity is usually given as a function of n, say $T(n)$, and similarly the storage complexity is also a function of n, say $S(n)$. If no preliminary information is available about the list (whether the numbers in it are ordered or not, etc.), then the algorithm must examine each number in order to decide which is the greatest. It follows from this that the time complexity of *any* algorithm finding the greatest of n numbers is *at least proportional to* n. It is expressed by the following notation:

$$T(n) = \Omega(n). \tag{2.2}$$

A rigorous definition of this and the other usual complexity notations can be found in the next subsection. Note that such statements can be made without having any algorithm for solving the given problem, thus such **lower bounds** are related rather to the problems themselves than to the concrete algorithms. Let us then examine an obvious algorithm for solving the maximum-search problem (the input list is denoted by $k_1, \ldots, k_n$):

```
FindMaximum(k_1, ..., k_n)
    M = k_1;                          // M: the greatest found so far
    for i = 2 to n do
        if k_i > M then
            M = k_i;
        endif
    endfor
    return M;
end
```

Let the time required by the assignment operator (=) be denoted by $T_=$, the time required to perform the comparison ($k_i > M$) by $T_>$ and the time needed to prepare for a cycle by T_{loop} (the time of an addition and a comparison).

The time T spent by the above algorithm can then be written as:

$$T = T_= + (n-1) \cdot T_> + m \cdot T_= + (n-1) \cdot T_{\text{loop}} \quad (m \leq n-1), \tag{2.3}$$

where m is number of situations when the variable M must be updated. The value of m can be $n-1$ in the **worst case** (that is when the numbers in the input list are in ascending order). Thus:

$$T \leq T_{=} + (n-1) \cdot (T_{>} + T_{=} + T_{\text{loop}}). \tag{2.4}$$

The conclusion is that the time spent by the algorithm is *at most proportional* to n. This is expressed by the following notation:

$$T(n) = O(n). \tag{2.5}$$

This, in fact, gives an **upper bound** on the complexity of the maximum-searching problem itself: it states that there exists an algorithm that can solve it in time proportional to n. The lower bound ($T(n) = \Omega(n)$) and the worst-case time complexity of the proposed algorithm ($T(n) = O(n)$) coincide in this case. Hence we say that the algorithm has an **optimal** (worst-case optimal) time complexity. The storage requirement of the algorithm is only one memory location that stores M, hence the storage complexity is independent of n, that is constant:

$$S(n) = O(1). \tag{2.6}$$

2.1.1 Complexity notations

In time complexity analysis usually not all operations are counted but rather only those ones that correspond to a *representative* set of operations called **key operations**, such as comparisons or assignments in the previous example. (The key operations should always be chosen carefully. In the case of matrix-matrix multiplication, as another example, the key operations are multiplications and additions.) The number of the actually performed key operations is expressed as a function of the input size. In doing so, one must ensure that the number (execution time) of the unaccounted-for operations is at most proportional to that of the key operations so that the running time of the algorithm is within a constant factor of the estimated time. In storage complexity analysis, the maximum amount of storage ever required during the execution of the algorithm is measured, also expressed as a function of the input size. However, instead of expressing these functions exactly, rather their **asymptotic behavior** is analyzed, that is when

the input size approaches infinity, and expressed by the following special notations.

The notations must be able to express both that the estimations are valid only within a constant factor and that they reflect the asymptotic behavior of the functions. The so-called "big-O" and related notations were originally suggested by Knuth [Knu76] and have since become standard complexity notations [PS85].

Let $f, g : N \mapsto R$ be two real-valued functions over the integer numbers. The notation

$$f = O(g) \tag{2.7}$$

denotes that we can find $c > 0$ and $n_0 \in N$ so that $f(n) \leq c \cdot g(n)$ if $n > n_0$, that is, the function f *grows at most at the rate* of g in asymptotic sense. In other words, g is an **upper bound** of f. For example, $n^2 + 3n + 1 = O(n^2) = O(n^3) = \ldots$ but $n^2 + 3n + 1 \neq O(n)$. The notation

$$f = \Omega(g) \tag{2.8}$$

denotes that we can find $c > 0$ and $n_0 \in N$ so that $f(n) \geq c \cdot g(n)$ if $n > n_0$, that is, f grows *at least at the rate* of g. In other words, g is a **lower bound** of f. For example, $n^2 + 3n + 1 = \Omega(n^2) = \Omega(n)$. Note that $f = \Omega(g)$ is equivalent with $g = O(f)$. Finally, the notation

$$f = \Theta(g) \tag{2.9}$$

denotes that we can find $c_1 > 0, c_2 > 0$ and $n_0 \in N$ so that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ if $n > n_0$, that is, f *grows exactly at the rate* of g. Note that $f = \Theta(g)$ is equivalent with $f = O(g)$ and $f = \Omega(g)$ at the same time.

An interesting property of complexity classification is that it is *maximum emphasizing* with respect to weighted sums of functions, in the following way. Let the function $H(n)$ be defined as the positively weighted sum of two functions that belong to different classes:

$$H(n) = a \cdot F(n) + b \cdot G(n) \quad (a, b > 0) \tag{2.10}$$

where

$$F(n) = O(f(n)), \quad G(n) = O(g(n)), \quad g(n) \neq O(f(n)), \tag{2.11}$$

that is, $G(n)$ belongs to a higher class than $F(n)$. Then their combination, $H(n)$, belongs to the higher class:

$$H(n) = O(g(n)), \quad H(n) \neq O(f(n)). \tag{2.12}$$

Similar statements can be made about Ω and Θ.

The main advantage of the notations introduced in this section is that statements can be formulated about the complexity of algorithms in a hardware-independent way.

2.1.2 Complexity of graphics algorithms

Having introduced the "big-O" the effectiveness of an algorithm can be formalized. An alternative interpretation of the notation is that $O(f(n))$ denotes the class of all functions that grow not faster than f as $n \to \infty$. It defines a nested sequence of function classes:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(a^n) \tag{2.13}$$

where the basis of the logarithm can be any number greater than one, since the change of the basis can be compensated by a constant factor. Note, however, that this is not true for the basis a of the power ($a > 1$).

Let the time complexity of an algorithm be $T(n)$. Then the smaller the smallest function class containing $T(n)$ is, the faster the algorithm is. The same is true for storage complexity (although this statement would require more preparation, it would be so similar to that of time complexity that it is left for the reader).

When analyzing an algorithm, the goal is always to find the *smallest upper bound*, but it is not always possible. When constructing an algorithm, the goal is always to reach the *tightest known lower bound* (that is to construct an optimal algorithm), but it is neither always possible.

In **algorithm theory**, an algorithm is "good" if $T(n) = O(n^k)$ for some finite k. These are called *polynomial* algorithms because their running time is at most proportional to a polynomial of the input size. A given computational problem is considered as *practically tractable* if a polynomial algorithm exists that computes it. The *practically non-tractable* problems are those for which no polynomial algorithm exists. Of course, these problems

can also be solved computationally, but the running time of the possible algorithms is at least $O(a^n)$, that is *exponentially* grows with the input size.

In computer graphics or generally in CAD, where in many cases real-time answers are expected by the user (interactive dialogs), the borderline between "good" and "bad" algorithms is drawn much lower. An algorithm with a time complexity of $O(n^{17})$, for example, can hardly be imagined as a part of a CAD system, since just duplicating the input size would cause the processing to require 2^{17} (more than 100,000) times the original time to perform the same task on the bigger input. Although there is no commonly accepted standard for distinguishing between acceptable and non-acceptable algorithms, the authors' opinion is that the practical borderline is somewhere about $O(n^2)$.

A further important question arises when estimating the effectiveness of graphics or generally, geometric algorithms: what should be considered as the **input size**? If, for example, triangles (polygons) are to be transformed from one coordinate system into another one, then the total number of vertices is a proper measure of the input size, since these shapes can be transformed by transforming its vertices. If n is the number of vertices then the complexity of the transformation is $O(n)$ since the vertices can be transformed independently. If n is the number of triangles then the complexity is the same since each triangle has the same number of (three) vertices. Generally the input size (problem size) is the number of (usually simple) similar objects to be processed.

If the triangles must be drawn onto the screen, then the more pixels they cover the more time is required to paint each triangle. In this case, the size of the input is better characterized by the number of pixels covered than by the total number of vertices, although the number of pixels covered is related rather to the **output size**. If the number of triangles is n and they cover p pixels altogether (counting overlappings) then the time complexity of drawing them onto the screen is $O(n + p)$ since each triangle must be first transformed (and projected) and then painted. If the running time of an algorithm depends not only on the size of the input but also on the size of the output, then it is called an **output sensitive algorithm**.

2.1.3 Average-case complexity

Sometimes the worst-case time and storage complexity of an algorithm is very bad, although the situations responsible for the worst cases occur very rarely compared to all the possible situations. In such cases, an **average-case** estimation can give a better characterization than the standard worst-case analysis. A certain probability distribution of the input data is assumed and then the expected time complexity is calculated. Average-case analysis is not as commonly used as worst-case analysis because of the following reasons:

- The worst-case complexity and the average-case complexity for any reasonable distribution of input data coincide in many cases (just as for the maximum-search algorithm outlined above).
- The probability distribution of the input data is usually not known. It makes the result of the analysis questionable.
- The calculation of the expected complexity involves hard mathematics, mainly integral calculus. Thus average-case analysis is usually not easy to perform.

Although one must accept the above arguments (especially the second one), the following argument puts average-case analysis into new light.

Consider the problem of computing the convex hull of a set of n distinct points in the plane. (The convex hull is the smallest convex set containing all the points. It is a convex polygon in the planar case with its vertices coming from the point set.) It is known [Dév93] that the lower bound of the time complexity of *any* algorithm that solves this problem is $\Omega(n \log n)$. Although there are many algorithms computing the convex hull in the optimal $O(n \log n)$ time (see Graham's pioneer work [Gra72], for example), let us now consider another algorithm having a worse worst-case but an optimal average-case time complexity. The algorithm is due to Jarvis [Jar73] and is known as "gift wrapping". Let the input points be denoted by:

$$p_1, \ldots, p_n. \tag{2.14}$$

The algorithm first searches for an extremal point in a given direction. This point can be that with the smallest x-coordinate, for example. Let

it be denoted by p_{i_1}. This point is definitely a vertex of the convex hull. Then a direction vector $\vec{d}$ is set so that the line having this direction and going through p_{i_1} is a supporting line of the convex hull, that is, it does not intersect its interior. With the above choice for p_{i_1}, the direction of $\vec{d}$ can be the direction pointing vertically downwards. The next vertex of the convex hull, p_{i_2}, can then be found by searching for that point $p \in \{p_1, \ldots, p_n\} \setminus p_{i_1}$ for which the angle between the direction of $\vec{d}$ and the direction of $\vec{p_{i_1}p}$ is minimal. The further vertices can be found in a very similar way by first setting $\vec{d}$ to $\vec{p_{i_1}p_{i_2}}$ and p_{i_2} playing the role of p_{i_1}, etc. The search continues until the first vertex, p_{i_1}, is discovered again. The output of the algorithm is a sequence of points:

$$p_{i_1}, \ldots, p_{i_m} \tag{2.15}$$

where $m \leq n$ is the size of the convex hull. The time complexity of the algorithm is:

$$T(n) = O(mn) \tag{2.16}$$

since finding the smallest "left bend" takes $O(n)$ time in each of the m main steps. Note that the algorithm is *output sensitive*. The maximal value of m is n, hence the worst-case time complexity of the algorithm is $O(n^2)$.

Let us now recall an early result in geometric probability, due to Rényi and Sulanke [RS63] (also in [GS88]): the average size of the convex hull of n random points independently and **uniformly distributed** in a triangle is:

$$\mathrm{E}[m] = O(\log n). \tag{2.17}$$

This implies that the average-case time complexity of the "gift wrapping" algorithm is:

$$\mathrm{E}[T(n)] = O(n \log n). \tag{2.18}$$

The situation is very interesting: the average-case complexity belongs to a lower function *class* than the worst-case complexity; the difference between the two cases cannot be expressed by a constant factor but rather it grows infinitely as n approaches infinity! What does this mean?

The n input objects of the algorithm can be considered as a *point* of a multi-dimensional *configuration space*, say K_n. In the case of the convex hull problem, for example, $K_n = R^{2n}$, since each planar point can be defined by two coordinates. In average-case analysis, each point of the configuration space is given a non-zero probability (density). Since there is no reason

for giving different probability to different points, a *uniform* distribution is assumed, that is, each point of K_n has the same probability (density). Of course, the configuration space K_n must be bounded in order to be able to give non-zero probability to the points. This is why Rényi and Sulanke chose a triangle, say T, to contain the points and K_n was $T \times T \times \ldots \times T = T^n$ in that case. Let the time spent by the algorithm on processing a given configuration $K \in K_n$ be denoted by $\tau(K)$. Then, because of uniform distribution, the expected time complexity can be calculated as:

$$\mathrm{E}\left[T(n)\right] = \int\limits_{K_n} \frac{1}{|K_n|}\tau(K)dK, \tag{2.19}$$

where $|\cdot|$ denotes volume. The asymptotic behavior of $\mathrm{E}[T(n)]$ (as $n \to \infty$) characterizes the algorithm in the expected case. It belongs to a function class, say $O(f(n))$. Let the smallest function class containing the worst-case time complexity $T(n)$ be denoted by $O(g(n))$. The interesting situation is when $O(f(n)) \neq O(g(n))$, as in the case of "gift wrapping".

One more observation is worth mentioning here. It is in connection with the maximum-emphasizing property of the "big-O" classification, which was shown earlier (section 2.1.1). The integral 2.19 is the continuous analogue of a weighted sum, where the infinitesimal probability $dK/|K_n|$ plays the role of the weights a, b in equations 2.10–2.12. How can it then happen that, although the weight is the same everywhere in K_n (analogous to $a = b$), the result function belongs to a lower class than the worst-case function which is inevitably present in the summation? The answer is that the ratio of the situations "responsible for the worst-case" complexity and all the possible situations *tends to zero* as n grows to infinity. (A more rigorous discussion is to appear in [Már94].)

2.2 Parallelization of algorithms

Parallelization is the application of several computing units running parallelly to increase the overall computing speed by distributing the computational burden between the parallel processors.

As we have seen, image synthesis means the generation of pixel colors approximating an image of the graphics primitives from a given point of view.

More precisely, the input of this image generation is a collection of graphics **primitives** which are put through a series of **operations** identified as transformations, clipping, visibility calculations, shading, pixel manipulations and frame buffer access, and produce the **pixel data** stored in the frame buffer as output.

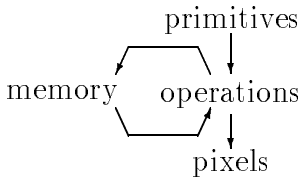

Figure 2.1: Key concepts of image generation

The key concepts of image synthesis (figure 2.1) — primitives, operations and pixels — form a simple structure which can make us think that operations represent a machine into which the primitives are fed one after the other and which generates the pixels, but this is not necessarily true. The final image depends not only on the individual primitives but also on their relationships used in visibility calculations and in shading. Thus, when a primitive is processed the "machine" of operations should be aware of the necessary properties of all other primitives to decide, for instance, whether this primitive is visible in a given pixel. This problem can be solved by two different approaches:

1. When some information is needed about a primitive it is input again into the machine of operations.

2. The image generation "machine" builds up an internal memory about the already processed primitives and their relationships, and uses this memory to answer questions referring to more than one primitives.

Although the second method requires redundant storage of information and therefore has additional memory requirements, it has several significant advantages over the first method. It does not require the model decomposition phase to run more times than needed, nor does it generate random order query requests to the model database. The records of the database can be

accessed once in their natural (most effective) order. The internal memory of the image synthesis machine can apply clever data structures optimized for its own algorithms, which makes its access much faster than the access of modeling database. When it comes to parallel implementation, these advantages become essential, thus only the second approach is worth considering as a possible candidate for parallelization. This decision, in fact, adds a fourth component to our key concepts, namely the internal **memory of primitive properties** (figure 2.1). The actual meaning of the "primitive properties" will be a function of the algorithm used in image synthesis.

When we think about realizing these algorithms by parallel hardware, the algorithms themselves must also be made suitable for parallel execution, which requires the decomposition of the original concept. This decomposition can either be accomplished functionally — that is, the algorithm is broken down into operations which can be executed parallelly — or be done in data space when the algorithm is broken down into similar parallel branches working with a smaller amount of data. Data decomposition can be further classified into input data decomposition where a parallel branch deals with only a portion of the input primitives, and output data decomposition where a parallel branch is responsible for producing the color of only a portion of the pixels. We might consider the parallelization of the memory of primitive properties as well, but that is not feasible because this memory is primarily responsible for storing information needed to resolve the dependence of primitives in visibility and shading calculations. Even if visibility, for instance, is calculated by several computing units, all of them need this information, thus it cannot be broken down into several independent parts. If separation is needed, then this has to be done by using redundant storage where each separate unit contains nearly the same information. Thus, the three basic approaches of making image synthesis algorithms parallel are:

1. **Functional decomposition** or **operation based parallelization** which allocates a different hardware unit for the different phases of the image synthesis. Since a primitive must go through every single phase, these units pass their results to the subsequent units forming a **pipeline structure** (figure 2.2). When we analyzed the phases needed for image synthesis (geometric manipulations, scan conversion and pixel operations etc.), we concluded that the algorithms, the ba-

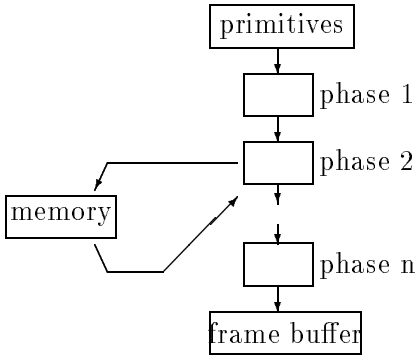

Figure 2.2: Pipeline architecture

sic data types and the speed requirements are very different in these phases, thus this pipeline architecture makes it possible to use hardware units optimized for the operations of the actual phase. The pipeline is really effective if the data are moving in a single direction in it. Thus, when a primitive is processed by a given phase, subsequent primitives can be dealt with by the previous phases and the previous primitives by the subsequent phases. This means that an n phase pipeline can deal with n number of primitives at the same time. If the different phases require approximately the same amount of time to process a single primitive, then the processing speed is increased by n times in the pipeline architecture. If the different phases need a different amount of time, then the slowest will determine the overall speed. Thus balancing the different phases is a crucial problem. This problem cannot be solved in an optimal way for all the different primitives because the "computational requirements" of a primitive in the different phases depend on different factors. Concerning geometric manipulations, the complexity of the calculation is determined by the number of vertices in a polygon mesh representation, while the complexity of pixel manipulations depends on the number of pixels covered by the projected polygon mesh. Thus, the pipeline can only be balanced for polygons of a given projected size. This optimal size must be determined by analyzing the "real applications".

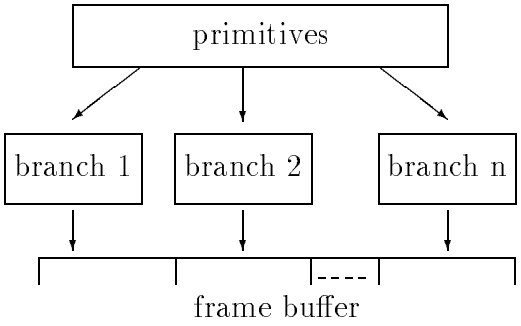

Figure 2.3: Image parallel architecture

2. **Image space** or **pixel oriented parallelization** allocates different hardware units for those calculations which generate the color of a given subset of pixels (figure 2.3). Since any primitive may affect any pixel, the parallel branches of computation must get all primitives. The different branches realize the very same algorithm including all steps of image generation. Algorithms which have computational complexity proportional to the number of pixels can benefit from this architecture, because each branch works on fewer pixels than the number of pixels in the frame buffer. Those algorithms, however, whose complexities are independent of the number of pixels (but usually proportional to some polynomial of the number of primitives), cannot be speeded up in this way, since the same algorithm should be carried out in each branch for all the different primitives, which require the same time as the calculation of all primitives by a single branch. Concerning only algorithms whose complexities depend on the number of pixels, the balancing of the different branches is also very important. Balancing means that from the same set of primitives the different phases generate the same number of pixels and the difficulty of calculating pixels is also evenly distributed between the branches. This can be achieved if the pixel space is partitioned in a way which orders adjacent pixels into different partitions, and the color of the pixels in the different partitions is generated by different branches of the parallel hardware.

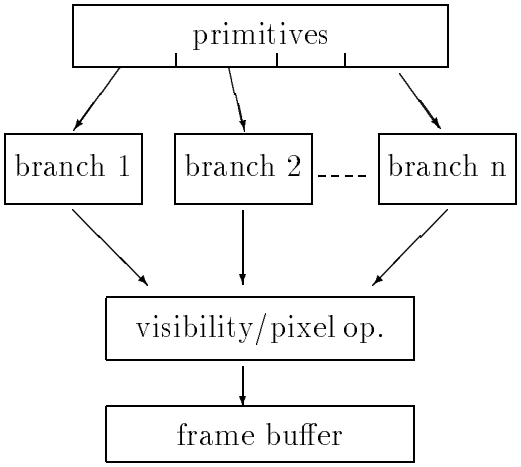

Figure 2.4: Object parallel architecture

3. **Object space** or **primitive oriented parallelization** allocates different hardware units for the calculation of different subsets of primitives (figure 2.4). The different branches now get only a portion of the original primitives and process them independently. However, the different branches must meet sometimes because of the following reasons: a) the image synthesis of the different primitives cannot be totally independent because their relative position is needed for visibility calculations, and the color of a primitive may affect the color of other primitives during shading; b) any primitive can affect the color of a pixel, thus, any parallel branch may try to determine the color of the same pixel, which generates a problem that must be resolved by visibility considerations. Consequently, the parallel branches must be bundled together into a single processing path for visibility, shading and frame buffer access operations. This common point can easily be a bottleneck. This is why this approach is not as widely accepted and used as the other two.

The three alternatives discussed above represent theoretically different approaches to build a parallel system for image synthesis. In practical applications, however, combination of the different approaches can be expected to provide the best solutions. This combination can be done in different ways, which lead to different heterogeneous architectures. The image parallel architecture, for instance, was said to be inefficient for those methods which are independent of the number of pixels. The first steps of image synthesis, including geometric manipulations, are typically such methods, thus it is worth doing them before the parallel branching of the computation usually by an initial pipeline. Inside the parallel branches, on the other hand, a sequence of different operations must be executed, which can be well done in a pipeline. The resulting architecture starts with a single pipeline which breaks down into several pipelines at some stage.

The analysis of the speed requirements in the different stages of a pipeline can lead to a different marriage between pipeline and image parallel architectures. Due to the fact that a primitive usually covers many pixels when projected, the time allowed for a single data element decreases drastically between geometric manipulations, scan conversion, pixel operations and frame buffer access. As far as scan conversion and pixel operations are concerned, their algorithms are usually simple and can be realized by a special digital hardware that can cope with the high speed requirements. The speed of the frame buffer access step, however, is limited by the access time of commercial memories, which is much less than needed by the performance of other stages. Thus, frame buffer access must be speeded up by parallelization, which leads to an architecture that is basically a pipeline but at some final stage it becomes an image parallel system.

2.3 Hardware realization of graphics algorithms

In this section the general aspects of the hardware realization of graphics, mostly scan conversion algorithms are discussed. Strictly speaking, hardware realization means a special, usually synchronous, digital network designed to determine the pixel data at the speed of its clock signal.

In order to describe the difficulty of the realization of a function as a combinational network by a given component set, the measure, called **combinational complexity** or combinational realizability complexity, is introduced:

Let f be a finite valued function on the domain of a subset of natural numbers $0, 1 \ldots N$. By definition, the combinational complexity of f is D if the minimal combinational realization of f, containing no feedback, consists of D devices from the respective component set.

One possible respective component set contains NAND gates only, another covers the functional elements of MSI and LSI circuits, including:

1. **Adders, combinational arithmetic/logic units (say 32 bits)** which can execute arithmetical operations.

2. **Multiplexers** which are usually responsible for the *then ... else ...* branching of conditional operations.

3. **Comparators** which generate logic values for *if* type decisions.

4. **Logic gates** which are used for simple logic operations and decisions.

5. **Encoders, decoders and memories of reasonable size (say 16 address bits)** which can realize arbitrary functions having small domains.

The requirement that the function should be integer valued and should have integer domain would appear to cause serious limitation from the point of view of computer graphics, but in fact it does not, since negative, fractional and floating point numbers are also represented in computers by binary combinations which can be interpreted as a positive integer code word in a binary number system.

2.3.1 Single-variate functions

Suppose that functions $f_1(k), f_2(k), \ldots, f_n(k)$ having integer domain have to be computed for the integers in an interval between k_s and k_e.

A computer program carrying out this task might look like this:

for $k = k_s$ **to** k_e **do**
    $F_1 = f_1(k)$; $F_2 = f_2(k)$; $\ldots F_n = f_n(k)$;
    write($k, F_1, F_2, \ldots F_n$);
endfor

If all the $f_i(k)$-s had low combinational complexity, then an independent combinational network would be devoted to each of them, and a separate hardware counter would generate the consecutive k values, making the hardware realization complete. This works for many pixel level operations, but usually fails for scan conversion algorithms due to their high combinational complexity.

Fortunately, there is a technique, called the **incremental concept**, which has proven successful for many scan conversion algorithms. According to the incremental concept, in many cases it is much simpler to calculate the value $f(k)$ from $f(k-1)$ instead of using only k, by a function of increment, $\mathcal{F}$:

$$f(k) = \mathcal{F}(f(k-1), k). \tag{2.20}$$

If this $\mathcal{F}$ has low complexity, it can be realized by a reasonably simple combinational network. In order to store the previous value $f(k-1)$, a register has to be allocated, and a counter has to be used to generate the consecutive values of k and stop the network after the last value has been computed. This consideration leads to an architecture of figure 2.5.

What happens if even $\mathcal{F}$ has too high complexity inhibiting its realization by an appropriate combinational circuit? The incremental concept might be applied to $\mathcal{F}$ as well, increasing the number of necessary temporary registers, but hopefully simplifying the combinatorial part, and that examination can also be repeated recursively if the result is not satisfactory. Finally, if this approach fails, we can turn to the simplification of the algorithm, or can select a different algorithm altogether.

Generally, the derivation of $\mathcal{F}$ requires heuristics, the careful examination and possibly the transformation of the mathematical definition or the computer program of $f(k)$. Systematic approaches, however, are available if $f(k)$ can be regarded as the restriction of a differentiable real function $f_r(r)$ to integers both in the domain and in the value set, since in this case

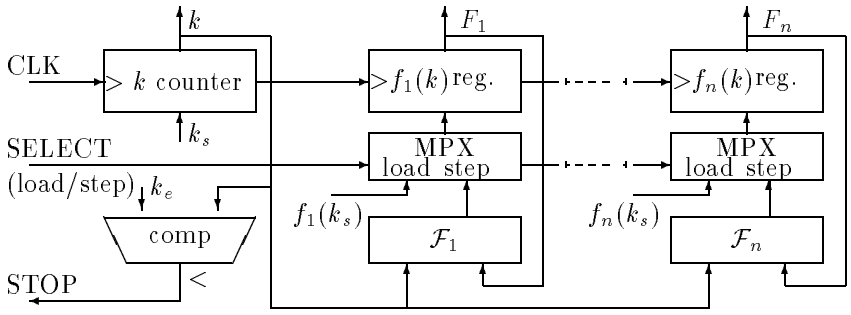

Figure 2.5: General architecture implementing the incremental concept

$f_r(k)$ can be approximated by **Taylor's series** around $f_r(k-1)$:

$$f_r(k) \approx f_r(k-1) + \frac{df_r}{dk}|_{k-1} \cdot \Delta k = f_r(k-1) + f_r'(k-1) \cdot 1 \qquad (2.21)$$

The only disappointing thing about this formula is that $f_r'(k-1)$ is usually not an integer, nor is $f_r(k-1)$, and it is not possible to ignore the fractional part, since the incremental formula will accumulate the error to an unacceptable degree. The values of $f_r(k)$ should rather be stored temporarily in a register as a real value, the computations should be carried out on real numbers, and the final $f(k)$ should be derived by finding the nearest integer from $f_r(k)$. The realization of floating point arithmetic is not at all simple; indeed its high combinational complexity makes it necessary to get rid of the floating point numbers. Non-integers, fortunately, can also be represented in **fixed point form** where the low b_F bits of the code word represent the fractional part, and the high b_I bits store the integer part. From a different point of view, a code word having binary code C represents the real number $C \cdot 2^{-b_F}$. Since fixed point fractional numbers can be handled in the same way as integers in addition, subtraction, comparison and selection (not in division or multiplication where they have to be shifted after the operation), and truncation is simple in the above component set, they do not need any extra calculation.

Let us devote some time to the determination of the length of the register needed to store $f_r(k)$. Concerning the integer part, $f(k)$, the truncation of

$f_r(k)$ may generate integers from 0 to N, requiring $b_I > \log_2 N$. The number of bits in the fractional part has to be set to avoid incorrect $f(k)$ calculations due to the cumulative error in $f_r(k)$. Since the maximum length of the iteration is N if $k_s = 0$ and $k_e = N$, and the maximum error introduced by a single step of the iteration is less than 2^{-b_F}, the cumulative error is maximum $N \cdot 2^{-b_F}$. Incorrect calculation of $f(k)$ is avoided if the cumulative error is less than 1:

$$N \cdot 2^{-b_F} < 1 \implies b_F > \log_2 N. \tag{2.22}$$

Since the results are expected in integer form they must be converted to integers at the final stage of the calculation. The Round function finding the nearest integer for a real number, however, has high combinational complexity. Fortunately, the Round function can be replaced by the Trunc function generating the integer part of a real number if 0.5 is added to the number to be converted. The implementation of the Trunc function poses no problem for fixed point representation, since just the bits corresponding to the fractional part must be neglected. This trick can generally be used if we want to get rid of the Round function.

The proposed approach is especially efficient if the functions to be calculated are linear, since that makes $f'(k-1) = \delta f$ a constant parameter, resulting in the network of figure 2.6. Note that the hardware consists of similar blocks, called interpolators, which are responsible for the generation of a single output variable.

The transformed program for linear functions is:

$F_1 = f_1(k_s) + 0.5$; $F_2 = f_2(k_s) + 0.5$; $\ldots F_n = f_n(k_s) + 0.5$;
$\delta f_1 = f_1'(k)$; $\delta f_2 = f_2'(k)$; $\ldots \delta f_n = f_n'(k)$;
for $k = k_s$ **to** k_e **do**
    write($k, \mathrm{Trunc}(F_1), \mathrm{Trunc}(F_2), \ldots, \mathrm{Trunc}(F_n)$);
    F_1 += δf_1; F_2 += δf_2; $\ldots F_n$ += δf_n;
endfor

The simplest example of the application of this method is the **DDA line generator** (DDA stands for Digital Differential Analyzer which means approximately the same as the incremental method in this context). For notational simplicity, suppose that the generator has to work for those

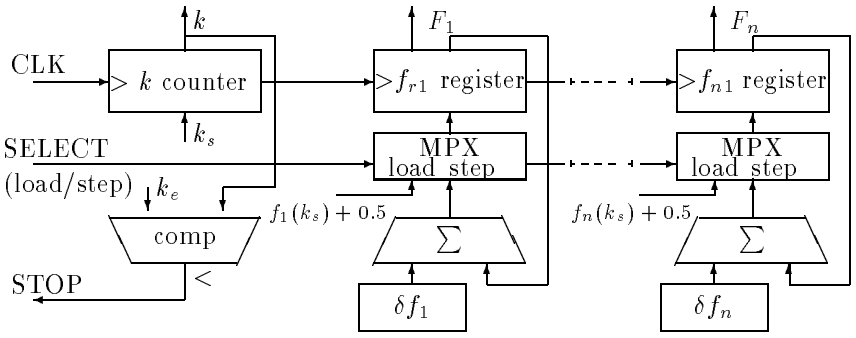

Figure 2.6: Hardware for linear functions

(x_1, y_1, x_2, y_2) line segments which satisfy:

$$x_1 \leq x_2, \qquad y_1 \leq y_2, \qquad x_2 - x_1 \geq y_2 - y_1. \tag{2.23}$$

Line segments of this type can be approximated by $n = x_2 - x_1 + 1$ pixels having consecutive x coordinates. The y coordinate of the pixels can be calculated from the equation of the line:

$$y = \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) + x_1 = m \cdot x + b. \tag{2.24}$$

Based on this formula, the algorithm needed to draw a line segment is:

for $x = x_1$ **to** x_2 **do**
    $y = \text{Round}(m \cdot x + b)$;
    write(x, y, color);
endfor

The function $f(x) = \text{Round}(m \cdot x + b)$ contains multiplication, non-integer addition, and the Round operation to find the nearest integer, resulting in a high value of combinational complexity. Fortunately the incremental concept can be applied since it can be regarded as the truncation of the real-valued, differentiable function:

$$f_r(x) = m \cdot x + b + 0.5 \tag{2.25}$$

Since f_r is differentiable, the incremental formula is:

$$f_r(x) = f_r(x) + f_r'(x-1) = f_r(x) + m. \tag{2.26}$$

The register storing $f_r(x)$ in fixed point format has to have more than $\log_2 N$ integer and more than $\log_2 N$ fractional bits, where N is the length of the longest line segment. For a display of 1280×1024 pixel resolution a 22 bit long register is required if it can be guaranteed by a previous clipping algorithm that no line segments will have coordinates outside the visible region of the display. From this point of view, clipping is not only important in that it speeds up the image synthesis by removing invisible parts, but it is also essential because it ensures the avoidance of overflows in scan conversion hardware working with fixed point numbers.

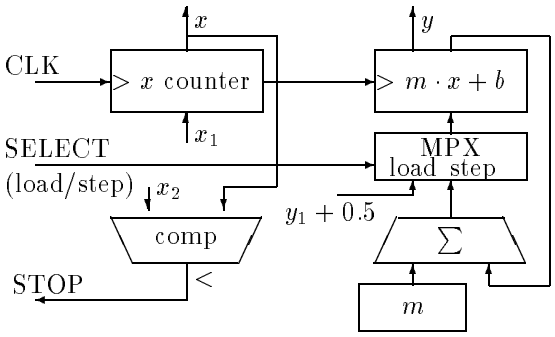

Figure 2.7: DDA line generator

The slope of the line $m = (y_2 - y_1)/(x_2 - x_1)$ has to be calculated only once and before inputting it into the hardware.

This example has confirmed that the hardware implementation of linear functions is a straightforward process, since it could remove all the multiplications and divisions from the inner cycle of the algorithm, and it requires them in the initialization phase only. For those linear functions where the fractional part is not relevant for the next phases of the image generation and $|\delta f| \leq 1$, the method can be even further optimized by reducing the computational burden of the initialization phase as well.

If the fractional part is not used later on, its only purpose is to determine when the integer part has to be incremented (or decremented) due to overflow caused by the cumulative increments δf. Since $\delta f \leq 1$, the maximum increase or decrease in the integer part must necessarily also be 1. From this perspective, the fractional part can also be regarded as an error value showing how accurate the integer approximation is. The error value, however, is not necessarily stored as a fractional number. Other representations, not requiring divisions during the initialization, can be found, as suggested by the method of **decision variables**.

Let the fractional part of f_r be fract and assume that the increment δf is generated as a rational number defined by a division whose elimination is the goal of this approach:

$$\delta f = \frac{K}{D}. \tag{2.27}$$

The overflow of fract happens when fract $+ \delta f > 1$. Let the new error variable be $E = 2D \cdot (\text{fract} - 1)$, requiring the following incremental formula for each cycle:

$$E(k) = 2D \cdot (\text{fract}(k) - 1) = 2D \cdot ([\text{fract}(k-1) + \delta f] - 1) = E(k-1) + 2K. \tag{2.28}$$

The recognition of overflow is also easy:

$$\text{fract}(k) \geq 1.0 \implies E(k) \geq 0 \tag{2.29}$$

If overflow happens, then the fractional part is decreased by one, since the bit which has the first positional significance overflowed to the integer part:

$$\text{fract}(k) = [\text{fract}(k-1) + \delta f] - 1 \implies E(k) = E(k-1) + 2(K - D). \tag{2.30}$$

Finally, the initial value of E comes from the fact that fract has to be initialized to 0.5, resulting in:

$$\text{fract}(0) = 0.5 \implies E(0) = -D. \tag{2.31}$$

Examining the formulae of E, we can conclude that they contain integer additions and comparisons, eliminating all the non-integer operations. Clearly, it is due to the multiplication by $2D$, where D compensates for the

fractional property of $\delta f = K/D$ and 2 compensates for the 0.5 initial value responsible for replacing Round by Trunc.

The first line generator of this type has been proposed by Bresenham [Bre65].

Having made the substitutions, $K = y_2 - y_1$ and $D = x_2 - x_1$, the code of the algorithm in the first octant of the plane is:

BresenhamLine(x_1, y_1, x_2, y_2)
  $\Delta x = x_2 - x_1$; $\Delta y = y_2 - y_1$;
  $E = -\Delta x$;
  $dE^+ = 2(\Delta y - \Delta x)$; $dE^- = 2\Delta y$;
  $y = y_1$;
  for $x = x_1$ **to** x_2 **do**
    if $E \leq 0$ **then** E += dE^-;
    else E += dE^+; y++;
    write$(x, y,$ color$)$;
  endfor

2.3.2 Multi-variate functions

The results of the previous section can be generalized to higher dimensions, but, for the purposes of this book, only the two-variate case has any practical importance, and this can be formulated as follows:

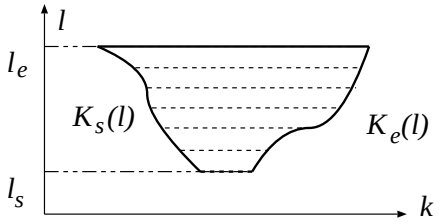

Figure 2.8: The domain of the two-variate functions

Let a set of two-variate functions be $f_1(k,l), f_2(k,l), \ldots, f_n(k,l)$ and suppose we have to compute them for domain points (figure 2.8):

$$S = \{(k,l) \mid l_s \leq l \leq l_e,\ K_s(l) \leq k \leq K_e(l)\}. \tag{2.32}$$

A possible program for this computation is:

for $l = l_s$ **to** l_e **do**
  for $k = K_s(l)$ **to** $K_e(l)$ **do**
    $F_1 = f_1(k,l)$; $F_2 = f_2(k,l)$; $\ldots F_n = f_n(k,l)$;
    write($k, l, F_1, F_2, \ldots F_n$);
  endfor
endfor

Functions f, K_s, K_e are assumed to be the truncations of real valued, differentiable functions to integers. Incremental formulae can be derived for these functions relying on Taylor's approximation:

$$f_r(k+1,l) \approx f_r(k,l) + \frac{\partial f_r(k,l)}{\partial k} \cdot 1 = f_r(k,l) + \delta f^k(k,l), \tag{2.33}$$

$$K_s(l+1) \approx K_s(l) + \frac{dK_s(l)}{dl} \cdot 1 = K_s(l) + \delta K_s(l), \tag{2.34}$$

$$K_e(l+1) \approx K_e(l) + \frac{dK_e(l)}{dl} \cdot 1 = K_e(l) + \delta K_e(l). \tag{2.35}$$

The increments of $f_r(k,l)$ along the boundary curve $K_s(l)$ is:

$$f_r(K_s(l+1), l+1) \approx f_r(K_s(l), l) + \frac{df_r(K_s(l), l)}{dl} = f_r(K_s(l), l) + \delta f^{l,s}(l). \tag{2.36}$$

These equations are used to transform the original program computing f_i-s:

$S = K_s(l_s) + 0.5$; $E = K_e(l_s) + 0.5$;
$F_1^s = f_1(K_s(l_s), l_s) + 0.5$; $\ldots F_n^s = f_n(K_s(l_s), l_s) + 0.5$;
for $l = l_s$ **to** l_e **do**
  $F_1 = F_1^s$; $F_2 = F_2^s$; $\ldots F_n = F_n^s$;
  for $k = \mathrm{Trunc}(S)$ **to** $\mathrm{Trunc}(E)$ **do**
    write($k, l, \mathrm{Trunc}(F_1), \mathrm{Trunc}(F_2), \ldots, \mathrm{Trunc}(F_n)$);
    F_1 += δf_1^k; F_2 += δf_2^k; $\ldots F_n$ += δf_n^k;
  endfor
  F_1^s += $\delta f_1^{l,s}$; F_2^s += $\delta f_2^{l,s}$; $\ldots F_n^s$ += $\delta f_n^{l,s}$;
  S += δK_s; E += δK_e;
endfor

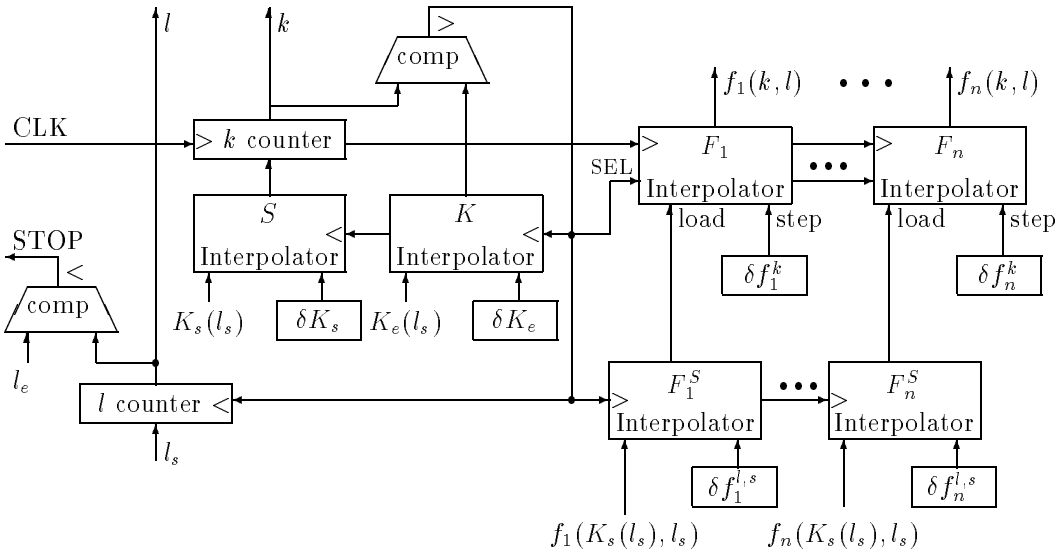

Figure 2.9: Hardware realization of two-variate functions

Concerning the hardware realization of this transformed program, a two level hierarchy of interpolators should be built. On the lower level interpolators have to be allocated for each F_i, which are initialized by a respective higher level interpolator generating F_i^s. The counters controlling the operation also form a two-level hierarchy. The higher level counter increments two additional interpolators, one for start position S, and one for end condition E, which, in turn, serve as start and stop control values for the lower level counter. Note that in the modified algorithm the longest path where the round-off errors can accumulate consists of

$$\max_l\{l - l_s + K_e(l) - K_s(l)\} \leq P_k + P_l$$

steps, where P_k and P_l are the size of the domain of k and l respectively. The minimum length of the fractional part can be calculated by:

$$b_F > \log_2(P_k + P_l). \tag{2.37}$$

A hardware implementation of the algorithm is shown in figure 2.9.

2.3.3 Alternating functions

Alternating functions have only two values in their value set, which alternates according to a selector function. They form an important set of non-differentiable functions in computer graphics, since pattern generators responsible for drawing line and tile patterns and characters fall into this category. Formally an alternating function is:

$$f(k) = F(s(k)), \qquad s(k) \in \{0, 1\}. \tag{2.38}$$

Function F may depend on other input parameters too, and it usually has small combinational complexity. The selector $s(k)$ may be periodic and is usually defined by a table. The hardware realization should, consequently, find the kth bit of the definition table to evaluate $f(k)$. A straightforward way to do that is to load the table into a shift register (or into a circular shift register if the selector is periodic) during initialization, and in each iteration select the first bit to provide $s(k)$ and shift the register to prepare for the next k value.

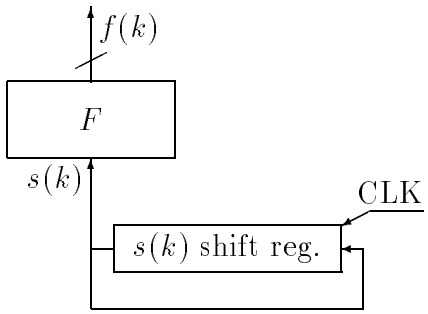

Figure 2.10: Hardware for alternating functions

Alternating functions can also be two-dimensional, for example, to generate tiles and characters. A possible architecture would require a horizontal and a vertical counter, and a shift register for each row of the pattern. The vertical counter selects the actual shift register, and the horizontal counter, incremented simultaneously with the register shift, determines when the vertical counter has to be incremented.