2. An Introduction to PDF

Adobe Systems Inc introduced PDF, the Portable Document Format, and their Acrobat software suite for interpreting PDF in 1993. The idea behind PDF, outlined by John Warnock in his 1991 paper 'The Camelot Project' [Warno91a], was simple — to provide a universal way to transfer graphically rich documents between heterogeneous operating systems and communication networks. In addition, a PDF file was to be viewable on any computer display or printer. In effect, PDF was designed to be the digital equivalent of a piece of paper, viewable by anyone, anywhere, regardless of the computer, software or display in use, and with the PDF file having the same appearance (subject to limitations of the equipment — colour would not be available on a monochrome printer, for example) wherever it was viewed.

PDF succeeded and has, over the last decade, has seen off competition from companies such as Novell (with *Envoy*) and Common Ground (with the imaginatively titled *Common Ground*) to become the *de facto* standard for the digital transfer of final form documents. PDF is rapidly replacing Adobe PostScript as the format for transferring documents to a print shop; high-end printers will now print PDFs directly, and Apple have used PDF as the basis for the *Quartz Graphics Engine*[Apple04a] in MacOS X.

2.1. PDF Structure

At an abstract level, a PDF file can be described as a collection of interconnected data structures which, as a whole, describe the structure and appearance of the document represented within the PDF. These data structures are analogous to those that would be built up within a typical computer program, and represent concepts such as a page, a collection of pages, fonts, the content of a page and images.

PDF is an open format, documented in the *PDF Reference Manual*[Adobe01a] and currently in its third edition. This book describes in detail all the objects used within a PDF, how they are arranged and connected to each other, and also how these objects can

be created from a number of smaller more general objects. It also details how the marks laid on the page interact with each other and how transformations can be applied to the operations, thereby defining PDF's graphics model.

A PDF file can be broken down into four major layers, this is illustrated in Figure 2.1 below. In the base layer, is the serialization syntax that forms the PDF file — a collection of ASCII characters that form tokens representing higher level objects, and allowing fast random access to them. On top of this are the base PDF types that form the backbone of a PDF. Built on these simple types are the more complex types mentioned above, representing objects such as pages. Finally, at the top, is the PDF graphics model, which describes how the drawing operations are imaged on the page and how they interact with one another.

Graphics Model
Complex Types
Simple Types
Serialization Syntax

Figure 2.1 — PDF Layer cake model

2.1.1. PDF Simple types

The easiest place to start explaining PDF is with the simple types that form the backbone of a PDF file. PDF documents are built up from eight basic types, five of which are atomic types, two compound types and finally the stream type. In addition to these types, PDF also supports the idea of "pointing to an object" in the same way that a programming language such as C or C++ does.

PDF's five atomic types represent the basic elements that are needed to define a PDF. These types, Booleans, Numbers, Strings,

Names, and the null object are mostly straight forward, representing standard computer science types. However, their implementation within PDF leads to some quirks that make them slightly different in operation from what might be expected in a conventional computer language.

2.1.1.1. Booleans

The simplest type used in PDF is the *boolean* type which has the same semantics as commonly encountered in computing circles. Its two possible states can represent true, or false.

2.1.1.2. Name objects

The *name* object is commonly used to label or name things. A name object consists of a sequence of characters (though as we shall see later the syntax of a PDF file restricts some of the characters that can be contained within a name). However although a name is made up of a sequence of characters these define the name and are not part of the name. Two name objects made up from the same sequence of characters are said to be identical and equal to each other (though they are distinct objects). Examples of name objects commonly seen within a PDF, might include `Page`, `Pages`, and `XObject`.

It is worth noting that it is common to see *name*s referred to with an initial / character, this is part of the syntax of PDF, where names are defined by a preceding slash character to distinguish them from operators (a throwback to PDF's PostScript heritage).

2.1.1.3. The Null Object

The *null* object is an object whose type and value is unequal to that of any other object. In fact there is only one *null* object. It is rarely seen within PDFs, but it sometimes does crop up when programatically generating PDF.

2.1.1.4. Numbers

PDF supports two representations of numbers: integer and real. Both are signed types, and are able to hold both positive and negative values. Although the size of *integer* objects is not defined in the PDF specification, it is limited by the structures used to represent them within the processing application. Within Adobe's own Acrobat viewer's *integer*s are limited to a 32-bit representation, meaning that the value of the object must be between -2147483648 (-2^{31}) and 2147483647 ($2^{31}-1$).

PDF implements real numbers in a fixed-point rather than a floating-point form. Although this is not an explicit rule, it is a strong recommendation. With most aspects, of PDF it is always wise to follow Acrobat's lead for implementation — divert from this and you run the risk of becoming incompatible with other PDF processors, either now or in the future. Acrobat represents real numbers in 32-bits, the most significant 16-bits representing the integer part (signed, so as to give a range between -32768 and 32767), and the bottom 16-bits storing the fractional part as the numerator to be divided by 65536. Thus, the number 1.5 would be represented as `0x00018000` with the top 16-bits (`0x0001`) representing the 1, and the bottom 16-bits (`0x8000` — 32768 decimal) representing the .5 as the fraction $\frac{32768}{65536}$. This leads to a precision of approximately 5 decimal places.

PDF does not differentiate between an *integer* and a *real* with a zero fractional part — serializing this latter quantity will lead to the same output, which when unserialized will create an integer object. Often PDF specifies that a *number* is required, rather than forcing it to be an *integer* or *real* object meaning that you can give either type of number object depending on what is required. An example might be the x,y co-ordinates of a graphical item. In future, the term *number* will be used to describe a value that can be either a *real* or an *integer*.

2.1.1.5. Strings

Strings are an interesting atomic type in PDF, becuase they are not quite atomic in the literal sense. Strings consist of a series of characters (actually unsigned bytes — giving 256 possible characters, though there are tricks to enable you to store the full Unicode range of characters, but that is beyond the scope of this document). However, within PDF a string is a string, it is not built up as a sequence of character objects like they are in traditional computer languages.

As with all PDF objects, PDF itself does not define any limits on the length of a string but if you are developing an application that processes PDF it is best to be compatible with what Adobe Acrobat does. In this case, strings are limited to 65535 bytes long.

2.1.1.6. Dictionaries

The *dictionary* object is the first of two composite objects, and is probably the most important object used in a PDF as it forms the backbone of the PDF structure. Most of the higher-level objects in a PDF (such as the one used to represent a page) are in fact dictionaries with an explicitly defined set of entries — in other words, a **Page** object is a dictionary that contains all the entries defined for a **Page** object.

Dictionaries are an unordered collection of objects paired with a *name* object, which forms a *key* for referencing the object within the collection. The keys must be unique; if a key appears twice in a dictionary then its value is undefined. Given that the keys are name objects (unlike in PostScript where any object could be used as a key), they are case-sensitive (see the rules above on equality of *name* objects).

The object referenced by a particular key (called the *value*) can be of any type, including another dictionary or it can be a reference to another object within the PDF file. A dictionary key-value pair with a *null* object as its value is equivalent to not defining an entry for that key. In other words, fetching an undefined key from a dictionary

would return the *null* object. Acrobat limits dictionaries to holding 4095 key-value pairs.

As previously mentioned, dictionaries form the backbone of PDF; the higher level objects are dictionaries with a defined set of entries. To this end it is the convention to add a `Type` entry to a dictionary which specifies the class of object that this dictionary represents. Further specification of the object's class is sometimes specified by a further `Subtype` key in the dictionary. An example would be the dictionary for representing a TrueType font. This has a `Type` of `Font` (as do the objects representing Type 1, Type 3 or any other type of font) and a `Subtype` of `TrueType`.

2.1.1.7. Arrays

While the dictionary is akin to the associative arrays found in languages such as *Awk*[Aho85a], and *PERL*[Wall00a], PDF also provides a more traditional array object similar to that found in languages, such as C. These arrays are a sequential list of objects that are indexed by their position in the array, the first object in the array being held at position zero. Arrays in PDF are heterogeneous rather than the strongly typed homogenous arrays found in C. This means that a PDF array can contain a mixture of strings, numbers, dictionaries or any other object, including other arrays. PDF supports only one-dimensional arrays, however it is possible to simulate a multi-dimensional array by having an array of arrays, nested to an arbitrary depth.

Again PDF doesn't specify an explicit limit on the number of elements within an array object, but to remain compatible with Acrobat one needs to respect its implementation limit of 8191 elements.

2.1.1.8. Streams

The final object type in PDF, is the *stream*. This object is used as a container for an arbitrary sequence of bytes. Unlike the *string* object which is also a sequence of bytes, a stream allows full random-access to the bytes in the same way that simple file I/O works. A stream is also of unlimited length — unlike the string where the length is limited by the architecture of the processing application. Typically, the stream object is used to contain data of an unknown (and potentially very large) size such as embedded images, and the descriptions of the page content. The meaning of a stream's content is defined by the context within which the stream is used.

A stream object consists of two parts, the data within the stream and a dictionary object which contains information about the data within the stream. The data within a stream may be encoded in a variety of different algorithms (known as *filters*), but this is only done when serializing the PDF. As far as the processing application is aware there is no difference between an encoded and an unencoded stream object.

The header dictionary contains information about the stream, and also any encodings applied to it when serialized. The only required entry is the `Length` which represents the size of the *decoded* stream data. It is perfectly legal for extra key-value pairs to be entered into the stream dictionary — they are ignored by any application that is not aware of them.

The stream object is unique within PDF as it is the only object that must be encoded as an indirect object, it therefore can not appear directly within an object such as an array or a dictionary.

2.1.1.9. Indirect references

PDF allows one to refer to objects. This allows for the sharing of objects. An example would be a font that appears on two separate pages, both **Page** objects can point to the same **Font** object rather than having to contain two separate but identical **Font** objects within

them. Any object type can be referenced indirectly.

The indirection mechanism is implemented by assigning each object an unique identification number which is used to refer to it. This unique identification number consists of two parts, an *object number* which labels the object and the *generation number* which shows what version of the object to use (this is to allow incremental updates to be made to a PDF file — something beyond the scope of this thesis). Both the object number and generation number are positive integers, however object number zero is not available for use as this is reserved.

2.1.2. PDF Syntax

PDF documents do not exist solely within a computer's memory; it must be possible to serialize a PDF into a format that can be stored on disk or transmitted over network connections. Due to its age and heritage, PDF has not yet adopted the modern method of serializing out to an XML representation (see next chapter), but instead it uses its own, predominantly ASCII, format that is based on Postscript notation.

Serialized PDF is broken into four segments, a *Header* which identifies the stream as a PDF, the *Body* which contains all the objects that make up the PDF, a *Cross-reference table* which speeds up access to objects within the PDF by providing byte offsets to the start of each object and a *Trailer* which enables a processing application to quickly find the Cross-reference table and other specialist objects within the file. Figure 2.2 contains a graphical representation of the structure of a PDF file.

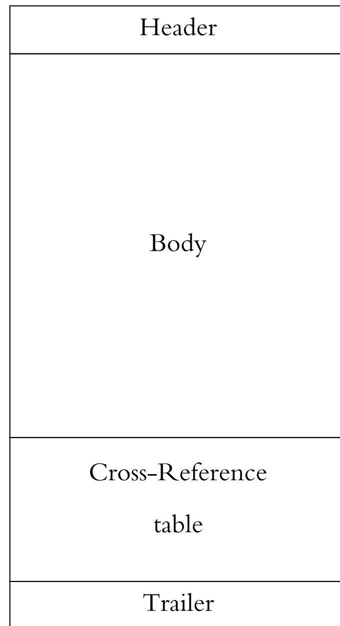

Figure 2.2 — Structure of a serialized PDF

A PDF file is a stream of 8-bit bytes. These bytes are grouped into tokens that can then be grouped to form the simple object types described above. Whilst a PDF can be described using only the printable subset of the ASCII standard (plus the white space characters — space, tab, carriage return and line feed), it is not restricted to this range and can make use of the full 8-bit byte range (for example, when encoding a stream object). Indeed, it is recommend in the interests of efficiency and file size that streams do make use of the full 8-bit range. This means that a PDF must be transported by a method that preserves the full 8-bit character range.

The tokens that represent the simple objects use the printable ASCII range as do all the names used as keys in the standard objects (represented in PDF as dictionaries).

The ASCII character set is divded into three groups or classes, *white-space* characters, *delimiter* characters and *regular* characters. These grouping determine how the stream is broken down into tokens, although within streams, strings and comments different rules apply. Table 2.1 shows how the ASCII character set is broken down into these groups.

Class	Hexadecimal	Decimal	Description
White-space	0x00	0	NULL character
	0x09	9	Tab (TAB)
	0x0a	10	Line Feed (LF)
	0x0c	12	Form Feed (FF)
	0x0d	13	Carriage Return (CR)
	0x20	32	Space (SP)
Delimiters	0x28	40	(
	0x29	41	)
	0x3c	60	<
	0x3e	63	>
	0x5b	91	[
	0x5d	93	]
	0x7b	123	{
	0x7d	125	}
	0x2f	47	/
	0x25	37	%
Regular	–	–	All other characters

Table 2.1 — Groupings of ASCII characters within PDF

White-space characters are used to separate tokens (such as a name followed by a number). All White-space characters are equivalent and multiple white-space characters are treated by PDF as being equivalent to a single white-space character, except when processing strings, streams and comments. In addition, the carriage return and line feed characters (sometimes knows as newline characters) are also treated as end-of-line markers (*EOL*). The combination of a carriage return

followed by a line feed (CR-LF — classic line ending of DOS/Windows systems) is treated as a single EOL marker. Most of the time, EOLs are treated identical to other white-space, but on some occasions they take on special meaning (often tokens need to be forced to appear at the beginning of a line).

Delimiter characters are used to delimit the start and end of strings, arrays, names and comments. The appearance of a delimiter character terminates the current entity, but is not included in the entity. For example, a string object starts with a '`(`' character and ends with a '`)`' but the actually string itself does not contain either of these delimiter characters. Also, when parsing an integer one should stop processing if a '`(`' character (for example) is encountered because the integer token is now complete, and start processing a string — though again neither the integer nor the string contain the '`(`'.

PDF also allows the addition of comments into the stream, these are denoted by the presence of a '`%`' character outside of a string or stream. The comment consists of all the characters between the '`%`' and the next EOL marker regardless of the class of characters between them. The comment has no semantics (with two exceptions, detailed below) and are effectively ignored by PDF, unlike in PostScript where they are used to structure the document (see [Adobe92a]), and are not guaranteed to be preserved by PDF consumers.

2.1.2.1. PDF Header

The Header in a PDF is there to provide a mechanism for identifying a byte stream as a serialized PDF file. It also contains the version number of the PDF specification that the PDF conforms to, although since PDF v1.4, it has been possibly to additionally override the version number by placing a `Version` entry into the document's catalog object (see later discussion). This header consists of a comment like the following:

```
%PDF-m.n
```

where `m` represents the major version number and `n` the minor. A PDF conforming to version 1.4 of the specification would therefore have a header line:

```
%PDF-1.4
```

Although not part of the PDF specification, it is often the case that there will be an additional comment in the PDF header containing at least four high ASCII characters (that is binary data greater than 128) if the PDF uses the full 8-bit byte range. This is to ensure that programs that try to analyze whether a document should be transferred as ASCII or binary (typically, mail and FTP clients), correctly determine that the file requires a binary transfer method.

2.1.2.2. PDF File Body

The Body part of the PDF is where all the actual objects are serialized. The rest of the file, contains information to ease access to these objects. The body consists of an ordered serialization of all the indirect PDF objects (which will include all the direct objects contained within these indirect objects — a direct object not contained within an indirect object is an impossibility) based on the object's object number.

The start and end of an indirect object is defined by the tokens `obj` and `endobj`. Preceding `obj` are the object number and generation number for this indirect object for this particular object, PDF is here showing its PostScript heritage by using postfix notation. A typical indirect object for the integer object representing the number 42 would look like:

```
21 0 obj
42
endobj
```

Calling this object out, within another object is done by use of the R token, again quoting the same object number and generation number used when defining the object with obj as above. So a reference to the above object, would look like the following:

```
21 0 R
```

Each object described in section 2.2.1, has its own serilization format

2.1.2.2.1. Boolean serialization

The Boolean object is serialized very simply, by emitting either the token `true` or the token `false` depending on the state of the object.

2.1.2.2.2. Null object serialization

Serializing the null object is even is even simpler than serializing a Boolean object, the serialized form, in every case, is the token `null`.

2.1.2.2.3. Name object serialization

The name object, as mentioned earlier, is serialized by preceding the name with a '/' character. It is possible to use any character in a name, however, delimiter or white-space characters must be escaped within the name by converting them to their hex code value preceded by a '#'. For example to serialize the name "A%B" you would serialize it as:

```
/A#23B
```

2.1.2.2.4. Numbers

Serilization of PDF's two number objects, integers and real are done by encoding the number as a decimal value, with real numbers

containing the fractional part after the after a decimal point. A real number can start with a decimal point (removing the leading zero) and both integers and reals can have an optional sign either a ‘+’ or a ‘−’ depending on whether the number is positive and negative (though of course, including the ‘+’ is inefficient). Some examples of various real and integer numbers:

```
12 42 0.1 56.0 -23 +44 -.002 0.0
```

This storage method for real objets leads us to a problem as it is possible for the serialization format to store numbers that can’t be represented within the standard Acrobat fixed-point format. An example of this would be the value 0.1 which would be stored in fixed point form as $\frac{6554}{65536}$, which is not quite 0.1. However, it is close enough that rounding errors will not be noticed on *current* display devices. Sensible rounding when converting real objects back to the serialized form will also stop $\frac{6554}{65536}$ being serialized as 0.100006 instead of 0.1, as one would expect. Also, when serializing a real object with a fractional part equal to zero the correct serialization is to output the integer part alone, ie 42 and not 42.0. This when unserialized will lead to an integer object being created, this will not cause a problem as integer objects can act like real object when necessary.

2.1.2.2.5. String objects

String objects contain an arbitrary sequence of 8-bit bytes normally this would represent readable text, though there is nothing to stop unprintable characters being included in a string. Therefore serialization must be able to preserve the full 8-bits.

Strings themselves are output surrounded by a ‘(’– ‘)’ pair. This means however that any occurrences of these characters within the string need to be escaped to stop premature end of parsing. The simplest method is to prefix the characters with a ‘\’ character, causing an opening bracket ‘(’ to be encoded as ‘\(’. The other option

if they appear as matched pairs (there are as many ‘(’ as ‘)’ within the string) then they can be encoded as is. This does make parsing slightly more harder as the grammar is no longer context free and a stack is required to make sure you correctly count the number of brackets parsed. As well as escaping brackets, the ‘\’ can also be used to escape other characters (white space characters, the backslash itself) in the same manner or by providing the ASCII character code in octal.

The other way to encode strings which can be more efficient if the string contains a lot of high ASCII characters is it encode it as a sequence of hexadecimal characters (the case of the A–F characters is ignored). These characters are encoded between ‘<’– ‘>’ pairs as in the following:

```
<4E5a6B213E4F>
```

If the last digit of the last hex pair in the string is zero it may optionally be omitted making the following two hex strings identical.

```
<F345A0>
<F345A>
```

2.1.2.2.6. Dictionary Object serialization

Dictionary objects when serialized are contained between a ‘<<’ token and a ‘<<’ (note these should not be confused with the hex string tokens described above). Inside the dictionary we find the key-value pairs serialized out, with the key (a name object) being serialized first. Note that as dictionaries are unordered collections the order that they appear in the serialized form is unknown.

An example dictionary containing the keys `Type`, `Subtype`, and `foo` would look like the following

```
<<
 /Type /Sample
 /Subtype /NoddyExample
 /Foo  42 0 R
>>
```

The `Type` and `Subtype` keys are paired with name objects, whilst the `Foo` key is paired with a reference to the indirect object 42 0.

2.1.2.2.7. Array object serialization

Like dictionaries, arrays contain direct serializations of its children, though as arrays are an ordered collection they must be serialized in order. Instead of the '<<'– '>>' wrappings of a dictionary, arrays use `[` and `]` to wrap the objects and there is no key provided, the index being its position in the array. A sample array contain a mixture of number, boolean and name objects is shown below:

```
[ 42 16 7 5.8 /Hello false 16 ]
```

2.1.2.2.8. Stream serialization

Stream objects are perhaps the most complex to serialize because as well as containing arbitrary binary data, they can also be compressed and encoded by a variety of algorithms. These algorithms are referred to as filters, an their presence must be mentioned in the stream's header dictionary, so the processing application can correctly decode and decompress the filtered stream to get back to the original binary data. The presence of the keys `Filter`, and `DecodeParams` in the stream's header dictionary indicate that the stream is encoded. Understanding of the encoding of streams with the different filters is not necessary to produce PDFs (though of course they will be less efficient than they could otherwise be), or for processing PDF with the various tools available (as they will handle the decoding and decompression of the filters).

The serialization format of the stream is to first serialize as a direct object the header dictionary. Next on a newline (after an EOL character), the token `stream` is emitted followed by another forced (and significant) newline. After this token, the stream data filtered according to the rules set in the header dictionary is emitted. Finally, on another newline the token `endstream` is emitted to end the stream data. A simple ASCII stream would look like this:

```
<<
  /Length 3
>>
stream
foo
endstream
```

This does cause problems because a PDF tokenizer does not know whether it is currently parsing a stream object or a dictionary until after it has finished parsing the dictionary and checking for the presence of the `stream` token.

2.1.2.3. Cross-reference table

The third segment of a PDF file is the Cross-Reference (or xref) table. The cross-reference table is there solely to speed up access into a PDF file, by allowing a PDF processing application, to jump to a specific object rather than having to parse the complete PDF file to find it. To this end, the cross-reference table consists of an ordered list of byte offsets into the file (with byte zero being the '`%`' part of the header) representing each object in turn. This could be inefficient if there are large gaps in object numbers, and so a method is provided for resetting the object number to a different point. This effectively breaks the cross-reference table down into subsections. It also eases the task of incremental updates, as you only need to add a subsection of the new cross-reference table in, and not repeat the entries for those objects that have not changed.

The Cross-reference section is delimited by the presence of the `xref` token on a line of its own (preceded and followed by an EOL marker). This is followed by one or more cross-reference subsections.

Each Cross-reference subsection begins with a line containing two integer numbers, the first is the object number of the first object in this subsection, and the second is the count of the number of objects listed in this subsection. So, a line like:

```
0 211
```

would indicate that the first object is object zero, and that there are 211 objects within this cross-reference subsection. Whilst this:

```
42 7
```

would indicate that there are 7 objects in this subsection beginning with object number 42. Following this line are several lines (one for each object described in the subsection) of the form

`nnnnnnnnnn ggggg x` *eol*

Where `nnnnnnnnnn` and `ggggg` are 10-digit and 5 digit decimal numbers respectively, these two numbers are separated by a single space. The end-of-line marker must be two bytes long, if the streams default EOL is a single byte (either a line feed or a carriage return) then it is preceded by a single space. This makes the length of each line in the cross-reference subsection exactly 20 bytes. The `x` parameter is used to distinguish whether this object number is in use (has an object attached to it) or is free (isn't connected to any object). Free objects normally happen when a PDF is updated by an application (perhaps removing an annotation from the PDF), but the application does not regenerate all the object numbers from scratch. For an in use object, the `x` parameter would be the '`n`' character, and the 10-digit number (represented by the `nnnnnnnnnn` in the example above) would contain the offset of the object within the stream. The second number (`ggggg`) contains the generation number of the object.

For free objects, the same syntax applies however the `x` parameter is now an '`f`' character. Instead of giving the byte offset of the object the `nnnnnnnnnn` parameter contains the number of the next free object, this forms a linked-list of free objects. To get into the list the first object (object zero) is always free and has a generation number of 65535. The last free object points back to object zero.

A typical Cross-reference table might look something like this example:

```
xref
0 6
0000000003 65535 f
0000000017 00000 n
0000000081 00000 n
0000000000 00007 f
0000000331 00000 n
0000000409 00000 n
```

2.1.2.4. Trailer

The trailer part of the PDF provides a PDF consumer with a defined access point that can be used to locate the cross-reference table and various important objects within a PDF, akin to the way a CPU will always start up by jumping to a specific location (the reset vector). Having checked the PDF header to confirm the stream is a PDF, a PDF consumer should read a PDF from its end (though it is possible to read it from the front if parsing a forward-only stream such as a TCP stream). The last line in the stream will contain the end of file marker `%%EOF`. Before this are two lines that contain the token `startxref` followed on the penultimate line of the stream by an offset to the start of the cross-reference table (that is a the byte containing the character '`x`' `xref` token). Finally the last part of the trailer (or the first part if working from the front of the stream), is the trailer dictionary and on the line preceding this the token `trailer`.

This makes the end of a stream containing a serialized PDF similar to the following:

```
trailer
<<
    Various key-value pairs
>>
startxref
Offset to cross-reference section
%%EOF
```

The **trailer** dictionary must be a direct object. and is used to inform a PDF consumer of where various essential PDF objects are stored (for example, the **catalog** object which forms the root of a PDF) and provide additional information about the file. The **trailer** object contains the following keys:

Key	Required/ Optional	Type	Description
Size	Required	integer	Contains the total number of entries in the cross reference table. Notionally equivalent to the highest object number in use plus one (because of the free object list pointer in position zero).
Prev	Required *(if file has more than one cross-reference section)*	integer	Byte offset within the stream to the beginning of the previous cross reference.
Root	Required	dictionary	Must be an indirect reference to the PDF's **catalog** object

The **trailer** can contain other keys, but these are beyond the scope of this thesis.

2.1.2.5. Incremental Updates and Linearization

To enable Acrobat features such as annotations (most visible in the tool that enables the user to add a yellow post-it note to a PDF) to not require a PDF to rebuilt from scratch. There is the option to add the extra objects, including any original objects that need to be replaced with updated versions, at the end of the file after the `%%EOF` token. Another cross-reference table and trailer dictionary are then added to allow the PDF to be parsed. When parsing a PDF consumer can follow the links in the **trailer** dictionaries back to find the first cross-reference table and then build up a composite table, by overlaying the tables in order.

Parsing a PDF generally requires the stream to be either parsed in its entirety or to be able to provide random access. When dealing with PDFs coming over network streams (such as a web server serving a PDF via HTTP), this would delay the appearance of the first page until the whole PDF had been downloaded. Adobe introduced the concept of *linearization* which makes use of the incremental update facility to enable fast access to the first page.

Briefly, within a linearized PDF the objects are ordered within the file, starting with the **catalog** and then all the objects for page 1, then those for page 2 and so on. Next the cross reference table is moved to the front of the PDF (by bending the incremental update specification). With these modifications, after a few kilobytes of data are parsed the consumer can start displaying the first page of the PDF, and whilst the user reads the first page the rest of the PDF can be downloaded in the background.

2.1.3. PDF Complex Types

A PDF Document is constructed from several complex types (or objects) that are arranged in a tree-like structure, with the **Catalog** at the root of the tree. Although often referred to as trees, there are so many interconnections and loops between objects within a PDF that they are more like graphs, however we shall use the term trees as per

convention. This tree structure can be split down into several subtress, the pages tree (which contain the pages of the document itself), the structure tree (which contains the logical structure of the document) and several other structures that are either deprecated or outside the scope of this document.

The most minimal PDF would only contain the pages tree. In this case, the **Catalog** object contains a reference to a **Pages** object which contains an array of references to either more **Pages** objects (allowing the pages tree to be balanced to speed up access as well as allowing for optimizations to be performed) or **Page** objects. Each **Page** object will contain a stream filled with the commands to image the content on that page. Also, both **Pages** and **Page** can contain a `Resources` key which points to the various resources used by that page (or in the case of **Pages**, all the pages that are descendants of that object).

The following pages contain descriptions of the various common objects that are used in PDF and are necessary to understanding the work presented later. These descriptions take the following format:

Description

This section contains a textual description of the object and its purpose.

Keys

This section details the keys in this object and what their purpose is.

Context

This section describes where this object can be found.

2.1.3.1. The Catalog Object

2.1.3.1.1. Description

The **Catalog** object forms the root of the PDF document, and is referenced by the `Root` entry in the trailer. As well as providing references to the objects defining the document content, logical structure, named destinations (for hyper-linking purposes) and other attributes, it also provides information on how the PDF should be displayed when initially loaded (whether it should display page thumbnails or the bookmarks, what zoom setting to use, etc).

2.1.3.1.2. Keys

Key	Required/ Optional	Type	Description
Type	Required	name	Always the name `Catalog`.
Pages	Required *(must be indirect)*	dictionary	A reference to a **Pages** object. This object is the root node of the *Page Tree*, and can be considered the main entry point to the content within a PDF.
Struct-Tree-Root Mark-Info	Optional	—	These keys reference objects that form the structure tree that can be added to a PDF as of version 1.4.

2.1.3.1.3. Context

The **Catalog** appears as the root of a PDF document.

2.1.3.2. The Pages Object

2.1.3.2.1. Description

The **Pages** object forms the nodes in what is commonly referred to as the *pages* tree. This tree forms a collection of **Page** and **Pages** objects such that when the tree is read in a depth-first manner, it will visit every **Page** object (which form the leaf nodes of the tree) in the order that they appear in the document — the first **Page** visited will be the representation of page 1 in the document, the second **Page** visited representing page 2 and so on.

In its simplest form, the pages tree consists of one **Pages** object which contains an array of **Page** objects; one for each page. However to optimize the performance of a PDF viewer it is common to find PDFs that have *balanced trees* representing the pages tree. This also allows resources (such as fonts etc) to be shared at a higher level by placing the **Resources** object reference on a **Pages** object rather than being duplicated on every descendant **Page** object.

2.1.3.2.2. Keys

Key	Required/ Optional	Type	Description
Type	Required	name	Always the name `Pages`.
Parent	Required *(must be indirect, not applicable in root node)*	dictionary	A reference to the **Pages** object directly above this in the pages tree.
Count	Required	integer	Count of the number of **Page** objects that are descendants of this object. This is equivalent to the count of all the leaf nodes.
Kids	Required	array	An array containing *indirect* references to the children of this node, which can be either **Page** or more **Pages** objects. This array can not be empty.

Note: Some of the keys in the **Page** object are also valid in this object, where by they are inherited by the descendants of this object although they will be overridden by the settings on the descendants. Details of which keys can be placed on a **Pages** as well as a **Page** object are given in the next section.

2.1.3.2.3. Context

Pages objects are valid either as the **Catalog**'s `Pages` entry, or as 'kids' of another **Pages** object.

2.1.3.3. The Page Object

2.1.3.3.1. Description

Page objects form the leaf nodes in the pages tree and represent a single page within the document. Each **Page** object contains details of any resources, such as any fonts or images, used solely by this page (the page may also use resources defined in ancestor **Pages** object as well), as well as pointing to a stream of PDF operators that are executed to image the page's content on to the display device.

PDF also allows the content to be represented as an array of multiple streams which allows the content to be segmented. PDF treats an array of streams in the same manner as if the data within the streams were concatenated together into one stream. This feature whilst not altering the effect of the PDF content, does ease the task of creating a PDF.

Some operators used in PDF (notably those imaging bitmap images) require the length of a piece of data to be provided before the data itself. Normally, this would require a two-pass generation phase, the first phase generates the data, whilst the second updates the length to the correct value. With PDFs segmentation of the content stream, we can avoid this dilemma. The PDF producer generates the code for the PDF up to the length field at which point it stops (*Stream 1*). It then outputs into a second stream (*Stream 2*), the data for the bitmap image. Once the image has been generated it can calculate the length of the stream and add this information into *Stream 1* along with any other data required. Finally, the producer can then either continue with the rest of the page by adding to *Stream 2* or more likely by starting a third stream. These streams would then be added into an array, so that the PDF consumer can read out the correct sequence of operators and data to regenerate the page. Other uses include stopping the PDF stream whenever a new resource (Font or bitmap image perhaps) is required, adding the object for that resource and then coming back to the page within a new stream

object.

Other information stored here includes the size of the page, including the specification of features for high-end phototypesetters such as bleed area and trim areas.

2.1.3.3.2. Keys

Key	**Required/ Optional**	**Type**	**Description**
Type	Required	name	Always the name `Page`.
Parent	Required *(must be indirect)*	dictionary	A reference to the **Pages** object directly above this in the pages tree.
Resources	Required	Dictionary	A **Resources** object that contains details of all the resources used within this Page that are not defined higher up the pages tree. An empty dictionary denotes that the page requires no resources, whilst the absence of this entry signifies that the resources are being inherited from an ancestor node within the pages tree.
MediaBox	Required	rectangle	A rectangle (PDF defines a rectangle to be an array containing four entries, the (x,y) co-ordinates of the lower-left and upper-right corners of the rectangle) expressed in the default user space units (normally points) representing the size of the page.

continued…

Key	Required/ Optional	Type	Description
Contents	Optional	stream or array	A *content stream* that can be used to image the page contents. As described above this can be either a single stream, or it can be split into multiple streams that are linked together by their order within an array. The streams must be divided between lexical tokens, but the division is usually unrelated to the page's content. PDF processing applications are not required to preserve the structure of the `Contents` array, they are free to join or split the stream as necessary. Note some Acrobat versions will not accept an empty `Contents` array. The omission of this page from a **Page** object signifies that the page is empty.

Note: Keys labelled in bold, are also valid in **Pages** objects, their properties are inherited by any descendant **Page** objects.

2.1.3.3.3. Context

Page objects are valid only as children of a **Pages** object.

2.1.3.4. The Resources Object

2.1.3.4.1. Description

Some of the PDF imaging operators used within a content stream require access to resources stored outside of the actual content stream itself. These include common resources such as fonts and bitmapped image data, through more unusual objects representing Patterns and Shadings to the bizarrely named `ExtGState` and `XObject`.

To allow these resources to be accessed from within the content stream, they are given a name. This name is unique to the page, a font called say `F1` on page 1 might not be the same font that is referred to as `F1` on the next page, although it might well be. Also these names do not bear any resemblance to the more common names of objects. So our font named `F1` will more usually be referred to as *Times-Roman* or *Helvetica-Bold* rather than `F1`. The names used to represent objects are normally kept as short as possible to reduce the size of the content stream so it is usual to find names of the form `F1`, `T1` or `Im1` rather than more descriptive names.

Within the **Resources** object the resources are broken down into categories: Graphics State, Colourspaces, Patterns, Shadings, External Objects, and Fonts. Additionally, it is common to find a `ProcSet` entry which describes the types of operators used within the content stream. This was added to tell the viewer which Postscript Procedure Sets (sets of procedures that enable the PDF content stream to be executed by a PostScript VM) needed to be sent to print this page. As of PDF 1.4 this entry is considered obsolete and PDF consumers should not consider it to be accurate though it is strongly recommended that PDF producers do continue to produce accurate `ProcSet` arrays. Table 2.1 describes the defined procedure set categories.

Category Name	Description
PDF	Painting and Graphics State operators
Text	Text operators
ImageB	Greyscale images or image masks operators
ImageC	Colour images operators
ImageI	Indexed Colour Images operators

Table 2.1 — `ProcSet` Categories

As the **Resources** object only exists within a **Page** object it is not required to have `Type` entry.

2.1.3.4.2. Keys

Key	Required/ Optional	Type	Description
ExtGState	Optional	Dictionary	This maps to **GraphicState** objects that contain information about the graphics state that can not be set by using the traditional PDF operators.
XObject	Optional	Dictionary	This maps names to **XObject** objects which represent various 'External Objects'
Font	Optional	dictionary	This dictionary maps font names to the **Font** objects describing each particular font.
ProcSet	Optional	array	An array made up of the name objects listed in table 2.XX above, detailing what class of PDF graphic operators are used within this page.

2.1.3.4.3. Context

A **Resources** object is valid only as a child of a **Page** object.

2.1.3.5. The Font Object

2.1.3.5.1. Description

PDF provides support for several types of fonts including both the TrueType and PostScript Type 1 standards as well as the new Open-Type standard. PDF describes the fonts in a document by the use of a **Font** object. Each **Font** object provides the PDF consumer with details of what the font is really called (font's name, e.g. Times-Roman), how the character codes used in the document map to the glyphs (the character encoding), the widths of each glyph, and a pointer to a **FontDescriptor** object which describes the data that makes up the glyph itself (or references an external system font).

PDF only allows 8-bits to be used to encode references to glyphs, allowing for 256 characters to be used. Although the font is capable of carrying descriptions for many more glyphs. Whilst 256 characters are enough to represent most Western European alphabets, for other alphabets (Kanji or Cyrillic for example) and pictographic languages such as Chinese this 256 limit is too small. PDF therefore provides a further type of font called CID fonts, which allow up to 65536 characters to be used. This is done by allowing bank-switching of the 256 character blocks to be paged in and out as necessary, in the same manner that memory handling was done on older computer systems.

For the purposes of this thesis, discussion of fonts can be limited to the simple case of external font references. Here the **Font** and **FontDescriptor** objects are not required to have the actual glyph data contained within them, simplifying their implementation considerably. It is worth noting that for the standard Adobe 14 fonts (Times, Helvetica and Courier in 4 weights, plus the Symbol Font and Zapf Dingbats), a more minimal entry is allowed. In addition, we'll only look in detail at Type 1 fonts, although only minimal differences are required for other types.

2.1.3.5.2. Keys

Key	Required/ Optional	Type	Description
Type	Required	Name	Always the name `Font`.
Subtype	Required	name	This name defines what type of font this object describes. For example, a type 1 font would have the value `Type1` here. Other possible values include `Type3`, and `TrueType` for their respective font types.
Name	Optional	name	This is sometimes seen in very old PDFs and is the name the font is referenced by. This entry was required only in PDF version 1.0. Unused by modern PDF consumers, so PDF documents are no longer limited to referencing a font by the name assigned here.
BaseFont	Required	name	For a Type 1 font, this contains the actual name of the font as it would be referred to within a Postscript interpreter. For TrueType, an algorithm documented in [Adobe01a] is used to construct an equivalent `BaseFont` entry (typically this grabs the Postscript name from the Truetype header or uses information on how the host system refers to the font). For Type 3 fonts, there is no base font name so this entry is superfluous.

FirstChar	Required	integer	This contains the value of the first character code for which there is an entry in the `Widths` array (see below). This usually corresponds to the first imageable character in the font. For the Adobe standard 14 fonts, this entry can be omitted unless you wish to override the default font program and replace it with your own.
LastChar	Required	integer	This contains the value of the last character code for which there is an entry in the `Widths` array (see below). This usually corresponds to the first imageable character in the font. For the Adobe standard 14 fonts, this entry can be omitted unless you wish to override the default font program and replace it with your own.

continued…

Key	Required/ Optional	Type	Description
Widths	Required	array	An array of widths for all the characters between `FirstChar` and `LastChar` (resulting in `LastChar–FirstChar+1` entries). Each element is the the width of the glyph whose character code is the index into the array plus `FirstChar`. If a character code is unused in this font then a special proxy entry is used whose value is equal to that of the `MissingWidth` entry in the font's **FontDescriptor**. Glyph Widths are measured in units, such that 1000 units in font space corresponds to one unit in text space (see the section on the PDF graphics state for a discussion of the different graphics spaces available). Not required for the Adobe Standard 14, and it is preferred that when the entry is present it is an indirect reference.
FontDescriptor	Required	dictionary	Must be a reference to a **FontDescriptor** object. Not required for the Adobe Standard 14 fonts.

2.1.3.5.3. Context

Font objects are valid only as children of a **Resources** object.

2.1.3.6. The FontDescriptor Object

2.1.3.6.1. Description

Whilst the **Font** object tells the PDF consumer about a font's existence, it is the **FontDescriptor** object that actually describes the font itself, specifying the metrics, attributes and even the code to draw the glyphs themselves if necessary.

The metrics stored within the **FontDescriptor** object are equivalent to those found within an Adobe Type 1 fonts AFM font metrics file, and include the height of the capital letters from the baseline, the height of ascenders and descenders, the angle of the slope of the font (specified as an angle counter-clockwise from the vertical) and the width of vertical stems in characters. There is generally a one to one mapping of AFM entries to entries in the **FontDescriptor**. There is also a `Flags` entry which is a bitfield (stored as an integer) describing some of the font's properties (is it fixed-width, serif or sans serif etc).

Also if the font is to be embedded within the PDF file (rather than being a reference to a external font, typically a system font) then a copy of the font program is included in a stream pointed too by either the `FontFile`, `FontFile2`, or `FontFile3` keys depending on the type of font being processed.

2.1.3.6.2. Keys

Key	Required/ Optional	Type	Description
Type	Required	name	Must be `FontDescriptor`.
FontName	Required	name	The Postscript name of the font, must be the same as `BaseFont` in the **Font** object
Flags	Required	integer	A bitfield describing certain properties of the font, see Table 2.3 for details on the meaning of each bit.
FontBBox	Required	rectange	A PDF rectangle describing the bounding box for the font expressed in the glyph's graphics space. Can be thought of as the smallest rectangle that each glyph can fit into, or more explicitly the union of the bounding box of each glyph.
ItalicAngle	Required	number	The angle of slant for the dominant vertical strokes of the glyph within a font expressed in degrees counter-clockwise from the vertical. As most fonts slope to the right this value is almost always negative, rather than expressing the slant as a number between 270 and 360 degrees.
Ascent	Required	number	The highest y co-ordinate reached by glyphs (taking the zero point to be the baseline) in this font, excluding accented characters.
Descent	Required	number	The lowest y co-ordinate reached by glyphs (taking the zero point to be the baseline) in this font. Usually negative.
Leading	Optional	number	The desired spacing from baseline to baseline of two consecutive lines of text.
CapHeight	Required	number	The y co-ordinate of the top of a flat capital letter glyph, measured from the baseline.

StemV	Required	number	The thickness, horizontally, of the glyphs dominant vertical stems in the font.
FontFile FontFile2 FontFile3	Required	stream	Stream containing the font program for the glyphs used within this font. The precise name of the key depends on the type of font being encoded. A Type 1 font will be contained within a `FontFile` key, a TrueType within `FontFile2` and all other types are encoded within `FontFile3`. A **FontDescriptor** must contain at most only one of these keys.

Bit Position	Name	Meaning if set
1	FixedPitch	All glyphs have the same width.
2	Serif	Set if the font is a Serif font, clear otherwise.
3	Symbolic	Font contains characters that are outside the standard Adobe Latin character set. If this flag is set then Nonsymbolic must be clear. One of the flags must be set. One of the flags (`Symbolic` or `Nonsymbolic`) must be set.
4	Script	Indicates the font has a cursive, handwriting type look and feel.
6	Nonsymbolic	Font does not contain characters outside the standard Adobe Latin character set. If this flag is set then `Symbolic` must be clear. One of the flags (`Symbolic` or `Nonsymbolic`) must be set.
7	Italic	The font is an italic font, that is the characters are slanted.
17	AllCap	The font does not contain lower case letters.
18	SmallCap	The font's lowercase letters are actually smaller size versions of the uppercase letters.

19	ForceBold	Determines whether bold fonts are drawn with extra pixels even at small point sizes (ie when both bold and roman varieties would have widths of 1 pixel).

Table 2.3 — Font Flags

2.1.3.6.3. Context

FontDescriptor objects are valid only as a child of a **Font** object.

2.1.3.7. The XObject

2.1.3.7.1. Description

The **XObject** represents *External Objects*, graphical objects whose definition is self-contained and outside a page's content stream. There are three main classes of **XObject**s representing images, PostScript fragments, and the bizarrely named FormXObject…

PDF represents **XObject**s as stream objects, with their content data stored within the stream and stream dictionary containing meta-data about the **XObject** including what class the **XObject** belongs to, and as necessary information on how to render the **XObject**. The stream's dictionary will contain a `Subtype` key which informs the PDF consumer as to the class of XObject that this XObject belongs to.

XObject's are self-contained and do not rely on the calling page's graphics state to set their properties (though they may have affine transformations applied to them by operators within the page's content stream) so can be re-used multiple times and on multiple pages. Also, as they are self-contained imaging an XObject does not alter the page's graphics state. In other words, the only effect adding or removing an XObject to page's content stream has is whether the XObject is imaged or not on the page, the rest of the page's content would remain identical.

Image **XObject**s are one method used to represent bitmap images within a PDF page (the other is to directly include the image data within the page content stream, but this precludes the use of sharing the image data over multiple pages). The bitmap image data is then included as the stream's data, and the stream dictionary contains entries defining the width and height of the image in samples, the colour space used in hte image, how many bits are allocated per channel etc. The `Subtype` key for an image is `Image`.

PostScript XObjects were added in PDF version 1.1 to allow a PDF to be used to represent PostScript documents completely

because at that time PostScript contained features (transparency for example) that were not representable directly by the PDF syntax. As most PDF consumers are unlikely to be able to parse PostScript, they will have no effect when the PDF is viewed on screen or printed to a non-PostScript compatible printer. The only effect of these XObjects is when the PDF is printed to a PostScript printers, during which the contents of the **XObject**'s stream is copied verbatim into the PostScript output in place of the call out to the XObject.

With PDF version 1.4, PDF now supports all the features available in the PostScript level three imaging model and so this construct is no longer required. Adobe warns that it is likely to be removed from future versions of PDF and so recommends against its use.

Form XObjects have absolutely nothing to do with the PDF's ability to handle forms (e.g for tax returns, expenses claims job applications etc.) They are based on the PostScript Form [Adobe99a], the name being changed from verison 1.2 to avoid confusion once form-filling capabilities were added to PDF. They are referred to as 'FormXObject' as if it were just one word.

The PostScript Form was a specialised PostScript procedure which when executed made no alterations to the programming environment outside itself, the end result being that the output of the Form could be cached by the PostScript interpreter, so that it could be rendered much faster.

In PDF, FormXObjects have a similar idea, they are a set of PDF drawing operators that can be called out at any point in a page's content stream to image their content on the page.

An **XObject** object representing a form contains details of any resources used by the FormXObject (in an identical **Resources** object to those on **Page**(**s**) objects). Also included is a bounding box which is the smallest rectangle that completely encompasses all the graphics drawn by the FormXObject, or smaller if you wish to clip the drawing.

For the purposes of this thesis, we can limit our understanding to that of just FormXObjects.

2.1.3.7.2. Keys

Key	Required/ Optional	Type	Description
Type	Required	name	Always `XObject`.
Subtype	Required	name	Must be a name with value `Form` when representing a FormXObject. Other XObjects (Images, PostScript) will have different subtypes specified here.
FormType	Required	Integer	Must be the value 1. Required in earlier versions of PDF, though later versions list it as optional. Although no defined reason for its existence is given in [Adobe01a], [Hibba04a] describes how the feature was added originally as they envisaged there might be many types of FormXObject (in the same way as there are different types of fonts). As it happened there was only ever the one type of FormXObject and so the requirement for its presence was dropped.
BBox	Required	rectangle	A PDF rectangle (specialized array, see earlier) representing the bounding box of the object described within, or a portion of that is to be displayed (excess imagery is clipped).
Resources	Optional (but strongly recommended)	dictionary	A **Resources** object (equivalent to those used in **Page** and **Pages** objects) specifying the resources used by this FormXObject.

Matrix	Optional	array	An affine transform matrix that can be applied to the FormXObject as it is being drawn the default is the identity matrix which has no effect on the appearance of the FormXObject.

2.1.3.7.3. Context

XObject objects are valid only as children of a **Resources** object under the `XObject` key dictionary.

2.1.4. The PDF Graphics Model

The PDF Graphics model is identical to the graphics model used by Postscript, and was later adapted for use in the SVG web graphics format. At its heart the graphics model works by allowing paths to be built up that can then be stroked (their outline drawn) or filled (the insides filled). Provision is also provided to draw text, or image bitmaps.

PDF uses a standard Cartesian co-ordinate system with the origin initially defined to be in the bottom-left corner and the axis moving in a positive direction to the right and upwards. PDF has many co-ordinate spaces, depending on what context is being used. The first of these is the *Device Space* which is equivalent to the co-ordinate space of the output device, so on a screen 1 pixel is $\frac{1}{72}$ inches (assuming the 72dpi screen, these days it is closer to 100dpi) wide whereas on a phototypesetter you'd be talking closer to $\frac{1}{2400}$ inches wide (2400dpi or even higher). As this changes from device to device, PDF works in its own co-ordinate space, called *User Space* where each pixel is $\frac{1}{72}$ inches or one point (in PDF and Postscript, it is taken that there are 72 points to the inch, rather than the correct 72.27 points to the inch) wide. Part of the the job of a PDF consumer when rasterizing a PDF is to convert drawing operations from this *User Space* to *Device Space* so that the graphic is rendered at the correct size on the page or screen. This translation from user space to device space can be completed by using an affine transform, that scales, rotates and translates the x,y co-ordinates in user space to the correct ones in device space.

Additionally, PDF graphic operations are all subjected to an affine transform, defined by the *current transformation matrix* or *CTM.* All affine transforms can be defined by a 3x3 matrix that is applied to each (`x,y`) co-ordinate pair. Each co-ordinate pair is converted to a 3x1 matrix (the extra cell being padded with a 1), this can then be multiplied by the 3x3 matrix. Although the matrix is 3x3 only 6 of

the actual values are used as shown in the following equation:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

which when multiplied out gives the relationships

$$x' = ax + cy + e$$

$$y' = bx + dy + f$$

With these equations, it is possible to show that one matrix can contain any combination of scales, translation, rotation, and skewing. As an example the matrix below will translate the origin (and with it any drawing done) 100 pixels to the right and 100 pixels down.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & -100 & 1 \end{bmatrix}$$

If we add these values into the equations we get by expanding the multiplication, we get:

$$x' = 1x + 0y + 100$$

$$y' = 0x + 1y + -100$$

which simplify out to:

$$x' = x + 100$$

$$y' = y - 100$$

A scale operation to make everything twice as big would be represented by the following matrix:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which leads to the following equations:

$$x' = 2x$$

$$y' = 2y$$

The matrices for all four transformations are shown below in their general form:

$$\underset{\text{scale}}{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}} \underset{\text{translate}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}} \underset{\text{rotate}}{\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}} \underset{\text{skew}}{\begin{bmatrix} 1 & \tan\beta & 0 \\ \tan\alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}$$

Note the angles α and β refer to the horizontal and vertical skew angle respectively.

Combining transformations is simply a matter of multiplying two or more transformation matrices together. However, care must be taken with the ordering of the multiplication as they will not have the same effect. A translate followed by a rotation, moves the origin and then alters the rotation of the axes, whereas the other way round the translation is done along the direction of the newly rotated axes.

As well as *Device Space* and *User Space*, PDF also uses several other ‘spaces’ where necessary. An example of this is with text imaging where the text is drawn in *Text Space*. This is done so that text can be transformed without altering the transformations applied to the rest of the page.

The Graphic operations to draw a glyph are actually written to work in *Font Space*, which is defined as a 1000x1000 pixel grid (for an Adobe Type 1 font at least), such that the glyph for the letter ‘M’ or the letter ‘W’ would almost completely fill this area — in the standard Helvetica font the ‘W’ character is 944 units wide. Typically, this *Font Space* is considered to be equivalent to a 1pt square box in *Text Space*. So a transformation matrix applying a scale by a factor of 1000 must be applied to the glyph instructions to convert them to *Text Space*. This scaling matrix is usually stored with the font

procedure themselves or is hard coded into the specification for the fonts themselves (as is the case with Type 1 fonts).

It is possible to call up the glyph at a size other than 1pt square (e.g. at 12pts for body text), but it is more common (especially in PDFs produced by Adobe's Distiller program) to find the glyph's drawn at 1pt in text space and a scaling transformation is used that transforms the glyph from *Text Space* to *User Space* whilst also setting the point size. This is analogous to the PostScript convention of using the `scalefont` operator to set the point size.

By the time the glyph drawing operations are finally imaged they'll have been transformed by no less than five transformation matrixes — in order font space to text space, text space to user space, the current transform matrix and finally from user space to device space.

In a similar manner to fonts, XObjects are drawn in their own graphics space and run through a transform matrix to position them in the correct space. XObjects are usually drawn relative to the origin, so positioning is done by transforming the origin to the correct place in the page. With Image XObjects, the are considered by the PDF graphics model to be a 1x1 pixel square, so as well as being translated to the correct position, they need to be scaled to the correct size.

2.1.4.1. PDF Drawing Operators

Drawing operations within a PDF content stream are serialized as a sequence of operators, that may or may not take additional parameters in a reverse polish notation. This means that the parameters for the operator are serialized before the operator itself, an example would be the `Tj` operator which images a piece of text. This operator takes a string object as its parameter which would lead to a callout like the following:

```
(Hello World) Tj
```

Whilst the content stream is a sequence of operators, it does form a loose hierarchy as some operators must be contained within other operator pairs. The top level of this hierarchy would be the *Page Description Level* at this level general operators can be issued that alter the graphics state, colour, or the text state (but *not* image text). From this level, one can move down into the *Text Object* level (enclosed within `BT` and `ET` operators), the *Path Object* level (moved into by an operator that starts a path, and out of when the path is stroked or filled), and into an inline image object (by the `BI/EI` operator pair). Finally, there are two objects that have an immediate return to the page description level, an external object (entered and left by the `Do` operator) which causes an XObject to be executed at the current point in the page and a *Shading Object* (the `sh` operator).

The advantage of viewing the content stream in this manner is that you can consider just the objects described above as having a position and graphics state attributes attached to them, and then ignore the operators at the *Page description level*. Indeed, Adobe Acrobat provides this level of interface both graphically by the *Touch-Up* plugin and programmatically by what is known as the *PDFEdit* API. However, it is worth remembering that until the content stream has been parsed operators in one object can and will have an effect on operators in another. So switching a path object and a text object around can alter the appearance on the page.

Several PDF operators take transformation matrices as a parameter, such as the `cm` operator which sets the CTM. As the last column of the matrix never changes, only six parameters need be supplied to an operator. For a given transformation matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

`cm` operator as an example) thus:

```
a b c d e f cm
```

2.1.4.2. Text operations

There are two types of text operators those that are responsible for imaging text (and can only appear within a text object) and those that alter the text state.

Text state operators allow you to set the character spacing (`Tc` operator), leading (`TL`), word spacing (`Tw`), horizontal scaling (`Tz`), the text rendering mode — whether it is filled or stroked (`Tr`), and the text rise or offset above the baseline (`Ts`). All these operators take a number (real or integer) as a parameter to the operator, though it should be noted that each of these values has a sensible default value applied at the beginning of each content stream. In addition, the `Tf` operator allows you to select the font and size of the text, although as already mentioned it is more usual for text to be scaled with the text matrix (set by the `Tm` operator) rather than explicitly stated here. Unlike the other text state parameters, the font must be explicitly stated as there is no default value.

The most common text imaging and positioning operators are `Tm` for setting the text matrix, `Tj` for imaging a string and `TJ` which takes an array for strings and horizontal displacement values for imaging several strings with widths different to the standard character displacement between them, this removes the need to constantly reposition the text matrix and so in turn reduces the size of the content stream.

2.1.4.3. Path Objects

Path objects are begun by either the `m` operator, which begins a new path and moves to the x,y co-ordinate specified by its parameters or the `re` operator that draws a rectangle (parameters `x` and `y` of the lower left corner and the rectangle's width and height). The path object is finished by the issuing of an operator to paint the path to the screen or by converting the path into a clipping shape (the `W`/`W*` operators). In terms of painting a path, it can either be stroked (`S` operator), filled (`F`), or be both stroked and filled (`B`). Depending on

which algorithm you wish to use for filling (either non-zero winding number rule or the even-odd rule), you can append an `*` to either the `F` or `B` operator to designate that the even-odd rule is to be used. You may also wish the path to be closed when stroked (a line is drawn back to the starting point from the last position) in which case the operator should be given in lower-case, `s` instead of `S` for example.

Common path construction operators include straight line drawing (the `l` operator) and Bézier curves (`c`, `y` and `v` depending on the positioning of control points).

General graphics state operators allow the setting of the properties of the fill and stroke type used including the colour, line width and line style (whether it is dashed or not).

2.2. Programming PDF

It is possible to extend the Adobe Acrobat viewer's functionality by creating a plug-in, a small piece of code that Acrobat loads in at startup. Acrobat provides these plug-ins with a collection of C APIs that allows the plug-in to manipulate open PDF document as well as interacting with the user via the Acrobat user interface.

Acrobat provides several layers of APIs at different levels of abstraction. At the base level, Acrobat provides the COS API, in which each PDF simple type is represented by an equivalent opaque data type. The Acrobat COS API then provides the programmer with a collection of functions that can manipulate the types as one would expect, e.g. getting and setting values from a dictionary or an array.

2.3. Summary

This chapter has given an overview of the PDF format. It has outlined the basic types from which PDF files are constructed and how these types are serialized. It then showed how these basic types are

used to construct the structures that build up a PDF, including the common objects that represent pages, fonts and other resources. It then considered the graphics model used by PDF when imaging graphics before finally looking briefly at the programming APIs available from the Adobe Acrobat viewing software..bp

Bibliography

Adobe92a.

Adobe, *PostScript Language Document Structuring Conventions*, Adobe Systems Inc (25 September 1992). Technical Note #5001

Adobe99a.

Adobe, *Adobe PostScript Language Reference Manual (Third Edition)*, Addison-Wesley (December, 1999). page 206

Adobe01a.

Adobe, *PDF Reference Manual version 1.4*, Adobe Systems Inc (December, 2001).

Aho85a.

Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, "Awk — A Pattern Scanning and Processing Language Programmer's Manual", Computing Science Technical Report No. 118, Bell Labs, Murray Hill, New Jersey (June 5, 1985).

Apple04a.

Apple, *Quartz*, Apple Computer Inc (2004). http://developer.apple.com/quartz/

Hibba04a.

Peter Hibbard, *Personal Communication with*, 2004.

Wall00a.

Larry Wall, Tom Christiansen, and Jon Orwant, *Programming Perl (Third Edition)*, O'Reilley UK (July 2000).

Warno91a.

John Warnock, *The Camelot Project*, Adobe Systems Inc (1991).