The Portable Document Format and pdfTEX

S. Lurp

April, 2025

Table of Contents

1 Introduction

The Portable Document Format (PDF) is a file format developed by the Adobe corporation in 1992 to render and display documents. It is a rich file format capable of displaying a diverse variety of documents, and its immense popularity has lead to its use around the world. Still today it is the leading format for displaying documents in a cross-platform manner. It has undergone various updates and standardizations, keeping it modern and usable.

TEX is a program and language for typesetting (generally academic) documents. Historically, it compiled to a `dvi` (device independent) file format, but a more modern TEX engine called pdfTEX was developed to compile to PDF. TEX includes a powerful macro-based programming layer, as well as a versatile typesetting engine. We assume basic knowledge of plain-TEX for this article (if you don't know what plain-TEX is, try reading the TEXbook by D. Knuth before this article).

Despite the multitude of literature on PDF as well as TEX, there exists little literature on pdfTEX. Specifically, there does not exist much literature on how to utilize pdfTEX-primitives to create PDF graphics. The pdfTEX manual lists primitives, but does not give explanations on how to use them, and instead assumes intimate familiarity with PDF. In this article we will both explain PDF as well as how to utilize pdfTEX primitives to create PDFs.

We give thanks to the resources which were invaluable for this article:

- The pdfTEX manual by Hàn Thế Thành and team;
- The PDF reference by Adobe;
- Petr Olšák's article on pdfTEX primitives;
- The TikZ and PGF manual, as well as PGF source code.

Structure of the article

This article will be split largely into two parts: an introduction to PDF, and an explanation of pdfTEX primitives. The first part will not cover the entirety of the PDF, and we will leave certain subjects for the second part as well.

I. The Portable Document Format

The Portable Document Format (henceforth, PDF) is a powerful format for displaying documents. In this section we will discuss certain features of the PDF, focusing on its structure. In the following section we will discuss how to utilize pdfTeX primitives to alter the output PDF.

1 Datatypes and Functions

1.1 Datatypes

PDF supports the following datatypes:

- booleans;
- integers;
- real numbers;
- strings;
- names;
- arrays;
- dictionaries;
- streams;
- the null object.

1.1.1 Booleans

Booleans are determined by the keywords `true` and `false`. Integers are written with or without a sign, and a decimal point turns a number into a real number.

1.1.2 Strings

There are two ways to write a string: enclosing characters in parentheses, or, as hexadecimal data wrapped in single angle brackets (`<...>`). So for example, `(hello world)` represents the string "hello world". So does `<68656C6C6F776F726C64>`. To add special characters (line feed, unbalanced parentheses, etc) you can add a backslash before (as one would normally).

1.1.3 Names

A name begins with a forward slash `/`, and may contain any characters except for whitespace and delimiter characters (brackets, parentheses and friends, forward slash, or a percent sign). You can add any non-null character to a name (including special characters) by preceding its hexadecimal code with `#`. So for example, the following are names:

`/name`, `/name*with_special&characters`, `/#28parentheses#29`

1.1.4 Arrays

Arrays are one-dimensional array objects, written in square brackets. They are delimited by whitespace, and can include any other object as a member. For example, the following is an array:

`[1 1.2 true (S. Lurp) /A_Name [true false]]`

As displayed in the above example, an array can contain an array as well, thus allowing multi-dimensional arrays.

1.1.5 Dictionaries

A dictionary is a mapping between name objects and instances of any datatype. A dictionary is enclosed in double angle brackets, like so:

```
<<
    /Type /A_Dictionary
    /Subtype /Example
    /IntItem 12
    /NumItem 1.2
    /StringItem (hello world)
    /ArrItem [12 1.2 (hello world)]
    /DictItem <<
        /Item1 true
        /Item2 false
    >>
>>
```

Dictionaries are of extreme importantce in PDFs. They will show up a lot later.

1.1.6 Streams

A stream is similar to a string, except for a few key differences:

(1) a string must be read in its entirety, while a stream may be read incrementally;

(2) a string has a maximum limit based on implementation, while a stream has no such limit (which is why larger data like images or pages are stored in streams).

A stream is structured as follows:

```
<< dictionary >>
stream
...
endstream
```

A stream must be an *indirect object*. The stream's dictionary which precedes the `stream...endstream` must have a `/Length` field which is equal to the byte length of the stream's content (`...`).

1.1.7 Null

The null object is an object referencable by the keyword `null`. It is not equal to any other object. An indirect object which references a non-existent object is equivalent to the null object. And giving a dictionary entry the value `null` is equivalent to omitting the entry.

1.1.8 Indirect Objects

Any object can be labeled as an *indirect object*. This allows other objects to reference it via a unique *object identifier*. The object identifier has two parts:

(1) a unique positive integer called the *object number*;

(2) a (not necessarily unique) non-negative integer called the *generation number*. For our purposes, these are always zero.

An indirect object is declared using the `obj` postfix operator. Preceding it is the object identifier. The object's value itself is written between the `obj` and `endobj` keywords. For example:

```
12 0 obj
    [1 2 3]
endobj
```

Creates an array indirect object of value `[1 2 3]`. Its object number is 12, and its generation number is 0. To reference this object, simply use the *indirect reference* `12 0 R`.
For example, to create an indirect stream object, you could do:

```
7 0 obj
    << /Length 8 0 R >>
stream
    BT
        /F1 12 Tf
        72 712 Td
        (A stream with an indirect length) Tj
    ET
endstream
endobj

8 0 obj
    77
endobj
```

This defines object `7 0` to be a stream object containing the above contents. Its length is an indirect reference to object `8 0`, whose value is 77.

1.2 Trees

A tree is a composite datatype made up of other datatypes. Its purpose is similar to a dictionary: it maps keys to values, but by different means. They vary in some ways:

(1) the keys in a tree are either strings (name trees) or numbers (number trees);

(2) the keys are ordered;

(3) the values associated with the keys may be of any type (including `null`);

(4) a tree can represent an arbitrarily large map, and can be read in parts, unlike a dictionary.

There are two types of trees: name trees and number trees. The difference between them is the datatype of their keys: name trees use strings as keys while number trees use numbers.

Every tree is constructed from nodes, which are dictionary objects. There are three kinds of nodes: a root node, intermediate nodes, and leaf nodes. The meaning of these are self-explanatory.

1.2.1 Number trees

A number tree node is a dictionary with the following fields:

Key	Type	Value
Kids	array	(Root and intermediate nodes only; present in root only if **Names** isn't) an array of indiret references to the children of this node (either intermediate or leaf nodes).
Names	array	(Root and leaf nodes; present in root only if **Kids** isn't) an array of the form $$[key_1\ value_1\ \ldots\ key_n\ value_n]$$ where each key_i is a string, and the $value_i$ is the associated value. The keys are sorted by lexical value as explained below.
Limits	array	(Intermediate and leaf nodes only) an array of two strings, specifying the lexically least and greatest keys included in the **Names** array of a leaf node, or in the case of an intermediate node, the **Names** of the children nodes.

For example, we may have the following tree which describes the grades of students, say:

```
1 0 obj     % root
<<
    /Kids
      [ 2 0 R
        3 0 R ]
>>
endobj

2 0 obj     % intermediate
<<
    /Limits [(Andrew) (Gordon)]
    /Kids
      [ 4 0 R
        5 0 R
        6 0 R ]
>>
endobj

3 0 obj     % intermediate
<<
    /Limits [(Howard) (Zack)]
    /Kids
      [ 7 0 R
        8 0 R ]
>>
endobj

4 0 obj     % leaf
<<
    /Limits [(Andrew) (Avery)]
    /Names
      [ (Andrew) 100
        (Avery) 80 ]
>>
endobj

5 0 obj     % leaf
```

```
<<
    /Limits [(Bob) (Dylan)]
    /Names
      [ (Bob) 90
        (Chris) 100
        (Drew) 100
        (Dylan) 60 ]
>>
endobj

6 0 obj     % leaf
<<
    /Limits [(Fred) (Gordon)]
    /Names
      [ (Fred) 50
        (Gordon) 85 ]
>>
endobj

7 0 obj     % leaf
<<
    /Limits [(Howard) (Howard)]
    /Names
      [ (Howard) 10 ]
>>
endobj

7 0 obj     % leaf
<<
    /Limits [(Zack) (Zack)]
    /Names
      [ (Zack) 70 ]
>>
endobj
```

1.2.2 Number trees

A number tree is similar to a name tree except that its keys are integers instead of strings, and are sorted in ascending numerical order. And instead of the key-value array being named **Names**, it is named **Nums**.

1.3 Functions

An important quote from the PDF reference:

> “*PDF is not a programming language, and a PDF file is not a program.*”

Despite this, PDF provides the ability to define certain kinds of functions. Though of course their use is limited and restricted.

All PDF functions are pure functions $\mathbb{R}^m \to \mathbb{R}^n$ (pure meaning they have no side-effects). Importantly, their inputs and outputs must be numbers, not just any PDF datatype. PDF functions must have a domain defined in their definition. If a function has a domain of, say, $[-1, 1]$ and is called with input 6, the input will be clipped to the domain; so the function will be called with input 1. Similarly some functions may define a range, and the output may be similarly clipped.

A function may be either a dictionary or a stream. A *function dictionary* refers either directly to the function (if it is a dictionary) or to the stream dictionary (if it is a stream). The dictionary must provide a **FunctionType** entry, which is one of $0, 2, 3, 4$. For a function of type $\mathbb{R}^m \to \mathbb{R}^n$, the function dictionary may have the following fields (in addition to fields specific to the function type).

Key	**Type**	**Value**
FunctionType	integer	(Required) the function type.
Domain	array	(Required) an array of $2m$ numbers. For $0 \le i \le m-1$, $\mathbf{Domain}_{2i}$ must be less than or equal to $\mathbf{Domain}_{2i+1}$. The domain of the function is $$\prod_{0 \le i \le m-1} [\mathbf{Domain}_{2i}, \mathbf{Domain}_{2i+1}]$$

Range	array	(Required for type 0 and 4 functions, otherwise required) an array of $2n$ numbers. Similar to the domain, for every $0 \leq i \leq n-1$, $\textbf{Range}_{2i} \leq \textbf{Range}_{2i+1}$. The range (codomain) of the function is $$\prod_{0\leq i\leq n-1} [\textbf{Range}_{2i}, \textbf{Range}_{2i+1}]$$

1.3.1 Type 0 (Sampled) Functions

Type 0 functions use a sequence of sampled values (which are contained in a stream) to approximate a function whose domain and range are both bounded. In addition to the fields already listed, the function dictionary of a type 0 function may include the following fields as well:

Key	Type	Value
Size	array	(Required) an array of m positive integers which specifies the number of samples in each input dimension.
BitsPerSample	integer	(Required) the number of bits used to represent each sample. Valid values are $1, 2, 4, 8, 16, 24, 32$.
Order	integer	(Optional) the order of interpolation between samples. Valid values are 1 and 3, specifying linear and cubic spline interpolation, respectively. The default value is 1.
Encode	array	(Optional) an array of $2m$ numbers specifying a linear mapping of input values into the domain of the function's sample table. The default value is $[0\ (\textbf{Size}_0-1)\ 0\ (\textbf{Size}_1-1)\ \ldots]$.
Decode	array	(Optional) an array of $2n$ numbers specifying a linear mapping of sample values into the range appropriate appropriate for the function's output values.

The dictionary may include other fields common to stream objects. Given an input dimension of m, we must have $\prod_{0\leq i<m} \textbf{Size}_i$ values in the stream. In order, these give the multi-dimensional array

$$g(0,\ldots,0), g(1,\ldots,0), \ldots, g(\textbf{Size}_0-1, 0, \ldots, 0), g(\textbf{Size}_0-1, 1, \ldots, 0), \ldots, g(\textbf{Size}_0-1, \ldots, \textbf{Size}_{m-1}-1)$$

We now describe how to use g to compute f, the type 0 function.

To explain how the function is calculated, we first define the following function:

$$\text{Interpolate}(x; x_0, x_1, y_0, y_1) = y_0 + \left((x - x_0) \cdot \frac{y_1 - y_0}{x_1 - x_0}\right)$$

this simply projects x onto the line between (x_0, y_0) and (x_1, y_1).

When a sampled function is called with input values $(x_0, \ldots, x_{m-1})$, the following steps are taken in order to compute the result:

(1) Each x_i is clipped to the domain:

$$x_i' = \min(\max(x_i, \textbf{Domain}_{2i}), \textbf{Domain}_{2i+1})$$

(2) The input value is then encoded:

$$e_i = \text{Interpolate}(x_i'; \textbf{Domain}_{2i}, \textbf{Domain}_{2i+1}, \textbf{Encode}_{2i}, \textbf{Encode}_{2i+1})$$

That is, given an input x, we project it onto the line whose endpoints are $(\textbf{Domain}_{2i}, \textbf{Encode}_{2i})$ and $(\textbf{Domain}_{2i+1}, \textbf{Encode}_{2i+1})$. The effect is that the lower end of the domain is mapped to $\textbf{Encode}_{2i}$ and the higher end is mapped to $\textbf{Encode}_{2i+1}$.

(3) Then the input value is clipped

$$e_i' = \min(\max(e_i, 0), \textbf{Size}_i - 1)$$

This gives us a real matrix $(e'_m, \dots, e'_{m-1})$ which is the encoding of the input vector.

(4) We can then use interpolation (of order **Order**) to compute $g(e') = r$.

(5) We interpolate r in order to decode it:

$$r'_j = \text{Interpolate}(r_j; 0, 2^{\textbf{BitsPerSample}} - 1, \textbf{Decode}_{2j}, \textbf{Decode}_{2j+1})$$

(6) And then we clip the output to the range:

$$y_j = \min(\max(r'_j, \textbf{Range}_{2j}), \textbf{Range}_{2j+1})$$

This gives us the output: $f(x) = y$.

So for example, suppose we have a sampled function $\mathbb{R}^2 \to \mathbb{R}$ whose domain is $[-1,1]^2$. The sampling contains 21 columns and 31 rows. So we must encode the input to $[0,20] \times [0,30]$, and then decode to $[-1,1]$. So the code will be

```
14 0 obj
<<
    /FunctionType 0
    /Domain [-1.0 1.0 -1.0 1.0]
    /Size [21 31]
    /Encode [0 20 0 30]
    /BitsPerSample 4
    /Range [-1.0 1.0]
    /Decode [-1.0 1.0]
    /Length ...
>>
stream
... 651 sampled values ...
endstream
endobj
```

1.3.2 Type 2 (Exponential Interpolation) Functions

Type 2 functions are functions $\mathbb{R} \to \mathbb{R}^n$. Exponential interpolation is given by the following parameters $c_0, c_1 \in \mathbb{R}^n$ and $N \in \mathbb{R}$. The value of the function at x is

$$f(x; c_0, c_1, n) = c_0 + (c_1 - c_0)x^N$$

The interpretation of the parameters is as follows: c_0, c_1 are the values of f at $x = 0$ and $x = 1$ respectively. N is the interpolation exponent, which dictates how the curve behaves. When $N = 1$ this is simply linear interpolation.

A type 2 function dictionary may include the following additional fields:

Key	Type	Value
C0	array	(Optional) an array of n numbers, defining the parameter c_0. Its default value is [0.0].
C1	array	(Optional) an array of n numbers, defining the parameter c_1. Its default value is [1.0].
N	number	(Required) the interpolation exponent.

The values of **Domain** must constrain x such that if N is not an integer, $x \geq 0$. And if N is negative $x \neq 0$.

1.3.3 Type 3 (Stitching) Functions

Type 3 functions define a stitching of k one-input functions together. Suppose you're given a sequence of functions $f_0, \dots, f_{k-1} \colon \mathbb{R} \to \mathbb{R}^n$ with domains $[d_{00}, d_{01}], \dots, [d_{k-1,0}, d_{k-1,1}]$. Now we define another vector of *bounds*, **Bounds** $= [b_0, \dots, b_{k-2}]$, so that if $b_{i-1} \leq x < b_i$ we want to input x into f_i (b_{-1} and b_{k-1} are the endpoints of the domain specified for f). But $[b_i, b_{i+1}]$ may not align with the domain of f_i ($[d_{i0}, d_{i1}]$), and we don't necessarily want it to anyway. So we now we want to linearly interpolate $[b_i, b_{i+1}]$ into $[d_{i0}, d_{i1}]$. We do so by specifying two points $e_{i0}, e_{i1} \in [d_{i0}, d_{i1}]$ which will be the new endpoints.

Explicitly, we have

(1) a new domain $[d_0, d_1]$;

(2) k one-input functions **Functions** $= [f_0, \ldots, f_{k-1}]$ each with a domain $[d_{i0}, d_{i1}]$;

(3) a vector of bounds **Bounds** $= [b_0, \ldots, b_{k-2}]$ (and define $b_{-1} = d_0$ and $b_{k-1} = d_1$);

(4) a vector of encoding values **Encode** $= [e_{00}, e_{01}, \ldots, e_{k-1,0}, e_{k-1,1}]$;

and given an input value x, if $b_{i-1} \leq x < b_i$ (the right inequality is weak if $i = k - 1$), then we compute $x' = \text{Interpolate}(x; b_{i-1}, b_i, e_{i0}, e_{i1})$. Then we output $f_i(x')$.

Of course, the ranges of each f_i must be compatible with the range specified for f (if specified).

Key	Type	Value
Functions	array	(Required) the array of k one-input functions to be stitched together. Each function must have the same output dimensionality n.
Bounds	array	(Required) the array of $k - 1$ numbers determining the bounds for which to map into each function.
Encode	array	(Required) the array of $2k$ numbers which determines the mapping of bounds into domains of each function.

Notice that if we have a function $f\colon \mathbb{R} \to \mathbb{R}^n$, and we want to compute $g(x) = f(1 - x)$. We can compute this by defining g as a stitching function, where the **Encode** array is [1 0].

1.3.4 Type 4 (PostScript Calculator) Functions

A type 4 function utilizes a small subset of PostScript code to compute values. The following PostScript operators can be used in type 4 functions:

- Arithemtic operators: **abs, cvi, floor, mod, sin, add, cvr, idiv, mul, sqrt, atan, div, ln, neg, sub, ceiling, exp, log, round, truncate, cos**
- Boolean and bitwise operators: **and, false, le, not, true, bitshift, ge, lt, or, xor, eq, gt, ne**
- Conditional operators: **if, ifelse**
- Stack operators: **copy, exch, pop, dup, index, roll**

The operand syntax for type 4 functions follows PDF conventions rather than PostScript ones. The entire code defining the function must be wrapped in curly braces `{...}`. Braces are also used to delimit expressions executed conditionally in **if** and **ifelse** operators.

I may later update this document to provide information on how to use PostScript operators (specifically in PDFs).

2 File Structure

2.1 File structure

A "canonical" PDF file (pdfTEX only creates canonical PDF files. PDF files can be updated to create non-canonical PDFs, but we will not deal with those) has the following structure:

(1) a one-line *header* which specifies the PDF version of the file;

(2) a *body* containing all the objects which make up the file;

(3) a *cross-reference stream* containing information regarding the indirect objects in the file;

(4) a *trailer* which specifies the location of the cross-reference stream.

For example, if we were to compile the following TEX file:

hello-world.tex

```
\pdfcompresslevel=0 % don't compress the PDF
\nopagenumbers
Hello world!
\bye
```

we'd get the following PDF:

hello-world.pdf

```
%PDF-1.5
%
3 0 obj
<<
/Length 66
>>
stream
BT
/F1 9.9626 Tf 91.925 759.927 Td [(Hello)-333(w)27(orld!)]TJ
ET

endstream
endobj
8 0 obj
<<
/Length1 1472
/Length2 9273
/Length3 0
/Length 10745
>>
stream
%!PS-AdobeFont-1.0: CMR10 003.002
%%Title: CMR10
%Version: 003.002
...
endstream
endobj
11 0 obj
<<
/Producer (pdfTeX-1.40.26)
...
>>
endobj
5 0 obj
<<
/Type /ObjStm
/N 7
/First 43
/Length 1150
>>
stream
2 0 1 106 7 168 9 649 4 878 6 1010 10 1062
% 2 0 obj
<<
/Type /Page
/Contents 3 0 R
/Resources 1 0 R
/MediaBox [0 0 595.276 841.89]
/Parent 6 0 R
>>
% 1 0 obj
<<
/Font << /F1 4 0 R >>
/ProcSet [ /PDF /Text ]
>>
% 7 0 obj
[ ... ]
% 9 0 obj
<<
```

hello-world.pdf

```
/Type /FontDescriptor
/FontName /IEQIOR+CMR10
/Flags 4
/FontBBox [-40 -250 1009 750]
/Ascent 694
/CapHeight 683
/Descent -194
/ItalicAngle 0
/StemV 69
/XHeight 431
/CharSet (/H/d/e/exclam/l/o/r/w)
/FontFile 8 0 R
>>
% 4 0 obj
<<
/Type /Font
/Subtype /Type1
/BaseFont /IEQIOR+CMR10
/FontDescriptor 9 0 R
/FirstChar 33
/LastChar 119
/Widths 7 0 R
>>
% 6 0 obj
<<
/Type /Pages
/Count 1
/Kids [2 0 R]
>>
% 10 0 obj
<<
/Type /Catalog
/Pages 6 0 R
>>

endstream
endobj
12 0 obj
<<
/Type /XRef
/Index [0 13]
/Size 13
/W [1 2 1]
/Root 10 0 R
/Info 11 0 R
/ID [<804F5DFAA39FA28E85E2211C48BC665E> <804F5DFAA39FA28E85E2211C48BC665E>]
/Length 52
>>
stream
...
endstream
endobj
startxref
12479
%%EOF
```

I have truncated the PDF to save space and increase readability, but we still end up with over a hundred lines. Oh well.

Let's go over this file:

hello-world.pdf

```
1 %PDF-1.5
```

this is the header, it specifies that this file conforms to PDF version 1.5.

Lines 3 through 96 comprise the body of the PDF file.

Lines 97 through 111 comprise the cross-reference table. This contains information on how to randomly-access any indirect object within the file. This information (in PDF-1.5) is stored in the cross-reference stream (lines 108–110). The cross-reference stream is preceded by its stream dictionary, which may have the following fields:

Key	Type	Value
Type	name	(Required) the type of PDF object that the dictionary describes. Must be **XRef**.
Size	integer	(Required; must not be indirect) the highest object number used in the file, plus one.
Root	dictionary	(Required; must be indirect) an indirect reference to the catalog of the PDF (see below).
Info	dictionary	(Optional; must be indirect) an indirect reference to the document's information dictionary.
ID	array	(Optional) an array of two byte-strings (hexadecimal strings) which uniquely identifies the file.
Index	array	(Optional) an array of two non-negative integers. The first is the first object number in the file; the second is the number of objects. The default value is then [0 **Size**].
W	array	(Required) an array of integers corresponding to the size of the fields in a single cross-reference entry. We won't develop this further.

The cross reference stream in this file is

hello-world.pdf

```
97  12 0 obj
98  <<
99  /Type /XRef
100 /Index [0 13]
101 /Size 13
102 /W [1 2 1]
103 /Root 10 0 R
104 /Info 11 0 R
105 /ID [<804F5DFAA39FA28E85E2211C48BC665E> <804F5DFAA39FA28E85E2211C48BC665E>]
106 /Length 52
107 >>
108 stream
109 ...
110 endstream
111 endobj
```

and the trailer, which just specifies the byte offset of the cross reference stream, is

hello-world.pdf

```
112 startxref
113 12479
114 %%EOF
```

2.2 Document structure

2.2.1 The document catalog

The **Root** field in the file's cross-reference stream is an indirect reference to the document's *catalog*. The document catalog stores information for objects in the body of the file which define the document's content, outline, etc. It also contains information on how the document should be displayed.

We see on line 103 that the **Root** of the cross-reference stream is object `10 0`. This is the document catalog:

hello-world.pdf

```
89 % 10 0 obj
90 <<
91 /Type /Catalog
92 /Pages 6 0 R
93 >>
```

We see that the document catalog is a dictionary, and here it only has two entries. But in general it can have more:

Key	Type	Value
Type	name	(Required) the type of PDF object the dictionary describes. Must be **Catalog**.
Pages	dictionary	(Required; must be indirect) the *page tree node* (see below) that is the root of the document's *page tree* (see below).
PageLabels	number tree	(Optional) a number tree defining the page labeling for the document. The keys in the tree are page indicies, and the values are *page label dictionaries* (see below).
Dests	dictionary	(Optional) A dictionary of names and corresponding destinations (see below when we discuss hyperlinks).
URI	dictionary	(Optional) A dictionary containing information for URI actions (see below when we discuss hyperlinks).

2.2.2 The page tree

The pages of a document are accessed through a structure known as the *page tree*. This defines the structure of the pages in a document. A page tree has two types of nodes: intermediate nodes and leaf nodes. The most simple structure would be a single root intermediate node which contains as children all the pages in the document as leaf nodes. This is not efficient, so instead the tree is kept balanced generally.

A page tree node (intermediate node) is a dictionary with the following fields:

Key	Type	Value
Type	name	(Required) the type of PDF object that this dictionary describes. Must be **Pages**.
Parent	dictionary	(Required except in root; must be indirect) a reference to the parent of the tree node.
Kids	array	(Required) an array of indirect references to the immediate children of the node.
Count	interger	(Required) the number of leaf nodes (page objects) that are descendants of this node.

Note that the page tree does not necessarily reflect the logical structure of the document (chapters, sections, etc.).

A page object is a leaf in the page tree. It is a dictionary with the fields listed below. Some of the fields (those which are listed as such) may be inherited by ancestor nodes in the page tree.

Key	Type	Value
Type	name	(Required) the type of PDF object that this dictionary describes. Must be **Page**.
Parent	dictionary	(Required; must be indirect) a reference to the parent of the page object in the page tree.

Resources	dictionary	(Required; inheritable) a dictionary containing any resources required by the page. Omitting this indicates that it should be inherited.
MediaBox	array	(Required; inheritable) an array of four numbers defining the boundaries of the page.
Contents	stream	(Optional) a *content stream* (see below) which contains the contents of the page.
Group	dictionary	(Optional) a *group attributes dictionary* specifying the attributes of the page's page group for use in transparency (see below).
Annots	array	(Optional) an array of *annotation dictionaries* representing annotations associated with the page (see below).

More fields exist, but we ignore them for the sake of brevity. In order to inherit an attribute, place it in a page tree node, and all page objects which are its descendants will inherit the attribute.

We see in our document's catalog that the page tree root is object `6 0`:

hello-world.pdf

```
% 6 0 obj
<<
/Type /Pages
/Count 1
/Kids [2 0 R]
>>
```

We see that there is a single page, whose object is `2 0`:

hello-world.pdf

```
% 2 0 obj
<<
/Type /Page
/Contents 3 0 R
/Resources 1 0 R
/MediaBox [0 0 595.276 841.89]
/Parent 6 0 R
>>
```

The contents of the page are in object `3 0`. This is a content stream, which is a stream with operators telling the renderer how to display the page:

hello-world.pdf

```
3 0 obj
<<
/Length 66
>>
stream
BT
/F1 9.9626 Tf 91.925 759.927 Td [(Hello)-333(w)27(orld!)]TJ
ET

endstream
endobj
```

and the resources of the page are in object `1 0`:

hello-world.pdf

```
% 1 0 obj
<<
/Font << /F1 4 0 R >>
/ProcSet [ /PDF /Text ]
```

hello-world.pdf

```
>>
```

The resources dictionary defines `/F1` to be a font which is a reference to object `4 0`, which is a dictionary defining how to access the font.

2.2.3 Resource Dictionaries

Operands supplied to operators in content streams may only be direct objects. This places a heavy restriction that must be somehow overcome. For this reason, a content stream has a *resource dictionary*, defined by the **Resources** entry associated with it, in one of the following ways:

- for a content stream that is the value of a page's **Contents** entry, the resource dictionary is named by the page's **Resources** entry.
- for other content streams, the stream dictionary's **Resources** specifies the resource dictionary.
- A form XObject (see below) may omit the **Resources** entry, in which case the resources are looked up in the **Resources** entry of the page it is used in.

The entries in the **Resources** dictionary are as follows (all fields are optional). All entries are explained in more depth in their respective sections below.

Key	Type	Value
ExtGState	dictionary	A dictionary that maps resource names to graphics state parameter dictionaries.
ColorSpace	dictionary	A dictionary that each resource names to either the name of a device-dependent color space or an array describing a color space (see below).
Pattern	dictionary	A dictionary that maps resource names to pattern objects (see below).
Shading	dictionary	A dictionary that maps resource names to shading dictionaries (see below).
XObject	dictionary	A dictionary that maps resource names to XObjects. (see below).
Font	dictionary	A dictionary that maps resource names to font dictionaries.
ProcSet	array	An array of predefined procedure set names.
Properties	dictionary	A dictionary that maps resource names to property list dictionaries for marked content.

Each field is explained in more depth later in this article.

3 Graphics

PDF provides support for drawing and including graphics. In this section we will cover the code for doing so, and later on we will discuss how to interact with this code through pdfTEX.

PDF inherits the postfix syntax from PostScript. This inheritance is entirely syntactical, as PDF does not support the concept of an argument stack or other features PostScript provides.

PDF defines the following graphics objects for use within content streams:

- *path objects* are arbitrary shapes made up of straight lines, rectangles, and cubic Bézier curves. A path object ends with painting operators which indicate whether the path is opened or closed, stroked, filled, etc.
- *text objects* consist of one or more character strings that identify sequences of glyphs to be painted. It can also be stroked, filled, or used as a clipping boundary.
- *external objects* (XObjects) are objects defined outside of the content stream, but can be referenced from within the content stream through use of the stream's **Resources**. There are different kinds of XObjects:
    - *image XObjects* define a rectangular array of color samples to be painted;
    - *form XObjects* define an entire content stream to be treated as a single graphics object;
    - *reference XObjects* are a type of form XObject used to import content from one PDF into another;

  - *group XObjects* are a type of form XObject used to group graphical elements together (e.g. for use in the transparency model, which uses *transparency group XObjects*).

- *inline image objects* use a special syntax to express data for a small image directly within the content stream.

- *shading objects* describes a geometric shape whose color is an arbitrary function of position within the shape.

PDF 1.3 and early use an opaque imaging model, meaning that every object is painted in its entirety and at every point, only the object at the top has an effect on the color painted. PDF 1.4 and later use a transparent imaging model, meaning that objects may be specified to have a certain amount of transparency, so that objects underneath it may also affect the color painted. By default objects are painted as opaque.

3.1 Coordinate systems

Positions in the document are determined in terms of coordinates on a plane. A coordinate space is determined by the following properties relative to the current page:

- the location of the origin;

- the orientation of the x and y axes;

- the lengths of the units along each axis.

There are several coordinate spaces defined by the PDF, which will be described in this section. Transformations between coordinate spaces are done via affine transformations (transformations of the form $x \mapsto Ax + b$, where A is a matrix and b a vector).

The coordinate space which is native to a specific device is called its *device space*.

3.1.1 User space

To avoid the issues arising from using device-dependent coordinate spaces, PDF defines a device-independent coordinate system that remains the same relative to the current page no matter the medium in which it is displayed or printed. This is called the *user space* coordinate system.

The **CropBox** entry in a page dictionary specifies the rectangle of user space corresponding to the visible area on the output medium. The length of a unit along both the x and y axes is set by **UserUnit** (in PDF-1.6). If the entry is not supplied (or supported), the default value is 1/72th of an inch. **CropBox** defines the rectangular region for which the page is to be displayed in the infinite plane that is the user space.

The transformation from user to device space is defined by the CTM (current transformation matrix). This is stored in the PDF graphics state (to be dicussed below). A PDF content stream can modify user space by using the **cm** operator (coordinate transformation operator).

3.1.2 Other coordinate spaces

In addition to device and user space, PDF utilizes a variety of other coordinate systems:

- The coordinates of text are defined in *text space*. The translation from text space to user space is defined by a *text matrix* as well as several text-related parameters in the graphics state (see below).

- All sampled images are defined in *image space*. The transformation from image to user space is predefined and cannot be changed. All images are one-unit by one-unit in user space. To paint them, the CTM must be temporarily changed.

- A form XObject as a self-contained content stream is defined in a *form space*. When painted in another content stream, its space is transformed into user space using the *form matrix* which is defined in the form XObject.

- A pattern (which is content invoked repeatedly to tile an area) is defined in a space called *pattern space*. The transformation from pattern space to user space is defined in a *pattern matrix* contained in the pattern.

3.1.3 Transformation matrices

A transformation, as discussed previously, is an affine transformation of the form $x \mapsto Ax + b$ where A is a 2×2 matrix and b a vector of size 2. Suppose we want to transform (x, y) to (x', y') via such an affine transformation, then

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

or, we can write this as

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

The reason we add the final line of the matrix is to make it square. So if the current CTM is M and we'd like to transform it by the affine transformation described by matrix A, then we must change CTM to be $M \cdot A$. Thus if we are given an input coordinate X, then we first apply A and then M.

In the PDF reference, they take the convention of multiplying by row vectors on the left. So the affine transformation is represented by the tranpose of the matrix provided here. Nevertheless, the results are the same.

The affine transformation represented by

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

is represented in PDF code by $[a\ b\ c\ d\ e\ f]$.

3.2 Graphics State

When rendering a PDF, an internal state must be held by the application which determines the current state to be used when rendering graphics. The graphics state is initialized at the beginning of each page according to the table below.

Parameter	**Type**	**Value**
CTM	array	The current transformation matrix, which maps user space to device space. The CTM can be modified by use of the **cm** operator.
clipping path	(internal)	The current *clipping path*, which defines the boundary against which all output is to be cropped. The initial value is the boundary of the entire imageable portion of the page.
color space	name or array	The current *color space* which determines how color values are to be interpreted (e.g. RGB). There are two separate color spaces: one for stroking and one for filling. The initial value is **DeviceGray**.
color	(various)	The current color to be used when painting. The type and interpretation of the color depends on the current color space. There are two color parameters: one for stroking and one for filling. The initial value is black.
text state	(various)	A set of nine graphics state parameters that affect the painting of text (see below).
line width	number	The thickness, in user space units, of paths to be stroked. Initial value: 1.0.
line cap	integer	A code specifying the shape of endpoints for any stroked open paths (see below). Initial value: 0 (square caps).
line join	integer	A code specifying the shape of joints between connected segments of a stroked path. Initial value: 0 (mitered joints).
miter limit	number	The maximum length of mitered line joins for stroked paths (the length of the spikes when lines join at sharp angles). Initial value: 10.0.
dash pattern	array and number	A description of the dash pattern to be used when paths are stroked (see below). Initial value: a solid line.
blend mode	name or array	The current *blend mode* to be used in the transparent imaging model (see below). This parameter is reset to its initial value at the beginning of execution of a transparency group XObject. Initial value: **Normal**.

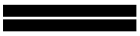
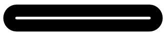
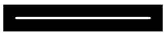
soft mask	dictionary or name	A *soft-mask* dictionary (see below), specifying the mask shape or opacity values to be used in the transparent imaging model, or **None**. This parameter is reset to its initial value at the beginning of execution of a transparency group XObject. Initial value: **None**.
alpha constant	number	The constant shape or constant opacity value to be used in the transparent imaging model. There are two alpha constant parameters: one for stroking and another for filling. This parameter is reset to its initial value at the beginning of execution of a transparency group XObject. Initial value: 1.0.
alpha source	boolean	A flag specifying whether the current soft mask and alpha constant parameters are to be interepreted as shape values (**true**) or opacity values (**false**). Initial value: **false**.

Some parameters are set with specific operators, while others are set by including a particular entry in a graphics state parameter dictionary. Some can be set either way. For example, the line width can be set using the **w** operator, or with the **LW** entry.

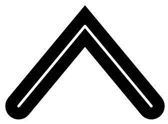
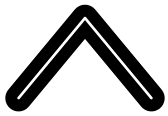
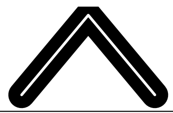
The *graphics state stack* allows for local changes to the graphics state, so you can change it without affecting things outside the current scope. The stack is a stack (LIFO — last in first out) data structure. The **q** operator pushes a copy of the graphics state onto the stack, while **Q** pops from the stack. Occurrences of **q** and **Q** must be balanced within a content stream.

3.2.1 Line caps

We demonstrate in the table below the three styles of line caps. These are the ends of open subpaths (and dashes) when they are stroked.

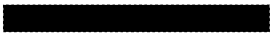
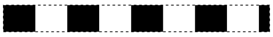
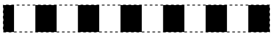
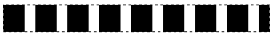
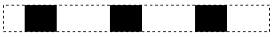
Style	Appearance	Description
0		*Butt cap*: the stroke is squared off at the endpoint of the path. There is no projection beyond the end of the path.
1		*Round cap*: the stroke is rounded off with semicircular ends on both sides of the stroke. The diameter of the capp is equal to the line width.
2		*Projecting square cap*: the stroke continues beyond the endpoint of the path for a distance equal to half the line width and is squared off.

3.2.2 Line Joins

Style	Appearance	Description
0		*Miter join*: the outer edges of the strokes for the segments are extended until they meet at an angle. If the angles meet at too sharp an angle (as defined by the miter limit parameter, though we won't go into this), a bevel join is used instead.
1		*Round join*: an arc of a circle with diameter equal to the line width is drawn around the point where the two segments meet.
2		*Bevel join*: ythe two segments are finished with butt caps.

3.2.3 Line dash pattern

The *line dash pattern* controls the pattern of dashes and gaps used by stroked paths. It is specified by a *dash array* and a *dash phase*. The dash array's elements are numbers that specify the lengths of alternating dashes and gaps (they must all be nonnegative and not all zero). The dash phase specifies the distance into the dash pattern at which to start the dash. When dashing begins, the elements of the dash array are cyclicly summed up, and when the sum equals the dash phase, the stroking of the phase begins.

Dash	Appearance	Description
[] 0		No dash
[3] 0		3 units on, 3 units off,. . .
[2] 1		1 on, 2 off, 2 on,. . .
[2 1] 0		2 on, 1 off, 2 on,. . .
[3 5] 6		2 off, 3 on, 5 off,. . .