

The Portable Document Format and pdfTeX

S. Lurp
April, 2025

Table of Contents

1	Introduction	4
I	The Portable Document Format	5
1	Datatypes and Functions	6
1.1	Datatypes	6
1.1.1	Booleans	6
1.1.2	Strings	6
1.1.3	Names	6
1.1.4	Arrays	6
1.1.5	Dictionaries	6
1.1.6	Streams	7
1.1.7	Null	7
1.1.8	Indirect Objects	7
1.2	Trees	8
1.2.1	Number trees	8
1.2.2	Number trees	9
1.3	Functions	9
1.3.1	Type 0 (Sampled) Functions	10
1.3.2	Type 2 (Exponential Interpolation) Functions	11
1.3.3	Type 3 (Stitching) Functions	11
1.3.4	Type 4 (PostScript Calculator) Functions	12
2	File Structure	12
2.1	File structure	12
2.2	Document structure	16
2.2.1	The document catalog	16
2.2.2	The page tree	16
2.2.3	The name dictionary	18
2.2.4	Resource Dictionaries	18
3	Graphics	19
3.1	Coordinate systems	19
3.1.1	User space	19
3.1.2	Other coordinate spaces	20
3.1.3	Transformation matrices	20
3.2	Graphics State	20
3.2.1	Line caps	21
3.2.2	Line Joins	22
3.2.3	Line dash pattern	22
3.2.4	Graphics state operators	22
3.3	Path construction and painting	24
3.3.1	Path construction operators	24
3.3.2	Path painting operators	25
3.3.3	Clipping path operators	27
3.4	Color Spaces	27

4	Patterns and Shadings	28
4.1	Tiling patterns	28
4.2	Shading Patterns	32
4.2.1	Type 1 (Function-Based) Shadings	33
4.2.2	Type 2 (Axial) Shadings	34
4.2.3	Type 3 (Radial) Shadings	35
4.2.4	The shading operator	36
5	External Objects	37
5.1	Images	37
5.1.1	Image XObjects	37
5.1.2	Inline images	39
5.2	Form XObjects	40
5.2.1	Group XObjects	41
6	Transparency	41
6.1	The idea	41
6.1.1	Compositing semi-transparent images	41
6.1.2	Blend functions	42
6.1.3	α -values and shape and opacity	45
6.1.4	Transparency groups	46
6.1.5	Soft masks	48
6.2	Specifying transparency in PDF	49
6.2.1	Soft mask dictionaries	49
6.2.2	Transparency group XObjects	49
6.3	An example	50
7	Text	51
7.1	Glyph structure	51
7.2	Text state	52
7.2.1	Character spacing	52
7.2.2	Word spacing	52
7.2.3	Horizontal scaling	53
7.2.4	Leading	53
7.2.5	Text rendering mode	53
7.2.6	Text rise	53
7.3	Text objects	53
7.3.1	Text space details	54
7.3.2	Text placement operators	54
7.3.3	Text-showing operators	55
8	Annotations	55
8.1	Border styles	56
8.2	Appearance streams	57
8.3	Text annotations	57
8.4	Link annotations	59
8.4.1	Destinations	60
8.4.2	Actions	60
II	pdfTeX	62
1	pdf literals	63
2	XObjects	65
2.1	Form XObjects	65
2.1.1	Shading	66
2.1.2	Transparency	67
2.2	Image XObjects	68
3	PDF Objects	69
3.1	Tiling patterns	69
3.2	PostScript functions	70
4	Annotations	71
4.1	Links and destinations	71

4.1.1	Destinations	72
5	Graphic State Stacks	72
5.1	Color stacks	72
5.2	The issue with transformations	73

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.

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

Despite the multitude of literature on PDF as well as T_EX, there exists little literature on pdfT_EX. Specifically, there does not exist much literature on how to utilize pdfT_EX-primitives to create PDF graphics. The pdfT_EX 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 pdfT_EX primitives to create PDFs.

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

- The pdfT_EX manual by Hàn Thê Thành and team;
- The PDF reference by Adobe;
- Petr Olšák's article on pdfT_EX primitives;
- The TikZ and PGF manual, as well as PGF source code.
- The paper <https://dl.acm.org/doi/10.1145/964965.808606> by Porter and Duff.

Structure of the article

This article will be split largely into two parts: an introduction to PDF, and an explanation of pdfT_EX primitives. The first part will not cover the entirety of the PDF standard, but parts I deem useful for pdfT_EX. The second part will similarly not cover the entirety of the pdfT_EX manual, and primitive usages will also be sometimes left incomplete (i.e. we will not discuss all the options possible for each primitive).

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:

```
1 <<
2   /Type /A_Dictionary
3   /Subtype /Example
4   /IntItem 12
5   /NumItem 1.2
6   /StringItem (hello world)
7   /ArrItem [12 1.2 (hello world)]
8   /DictItem <<
9     /Item1 true
```

```

10      /Item2 false
11    >>
12  >>

```

Dictionaries are of extreme importance 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:

```

1  << dictionary >>
2  stream
3  ...
4  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 (...).

Filters

A stream dictionary may specify a **Filter** field which is either a name or an array of names. These are used to decode the stream content between `stream` and `endstream`. If an array of filters is specified, they are composed on the stream contents in sequence.

We only discuss here **ASCIIHexDecode**, which decodes two characters, which represent a byte in hex, into a byte. For example, `0f` is decoded into a byte value of 15.

A `>` symbol must be placed at the end of the data.

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:

```

1 12 0 obj
2    [1 2 3]
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:

```

1 7 0 obj
2    << /Length 8 0 R >>
3  stream
4    BT
5      /F1 12 Tf
6      72 712 Td
7      (A stream with an indirect length) Tj
8    ET

```

```

9  endstream
10 endobj
11
12 8 0 obj
13     77
14 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 indirect 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 <div style="text-align: center;"> $[key_1 \ value_1 \ \dots \ key_n \ value_n]$ </div> <p>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.</p>
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 1 0 obj      % root
2  <<
3     /Kids
4     [ 2 0 R
5       3 0 R ]
6  >>
7 endobj
8
9 2 0 obj      % intermediate
10 <<
11    /Limits [(Andrew) (Gordon)]
12    /Kids
13    [ 4 0 R
14      5 0 R
15      6 0 R ]
16  >>
17 endobj
18

```



```

19 3 0 obj      % intermediate
20 <<
21   /Limits [(Howard) (Zack)]
22   /Kids
23     [ 7 0 R
24       8 0 R ]
25 >>
26 endobj
27
28 4 0 obj      % leaf
29 <<
30   /Limits [(Andrew) (Avery)]
31   /Names
32     [ (Andrew) 100
33       (Avery) 80 ]
34 >>
35 endobj
36
37 5 0 obj      % leaf
38 <<
39   /Limits [(Bob) (Dylan)]
40   /Names
41     [ (Bob) 90
42       (Chris) 100
43       (Drew) 100
44       (Dylan) 60 ]
45 >>
46 endobj
47
48 6 0 obj      % leaf
49 <<
50   /Limits [(Fred) (Gordon)]
51   /Names
52     [ (Fred) 50
53       (Gordon) 85 ]
54 >>
55 endobj
56
57 7 0 obj      % leaf
58 <<
59   /Limits [(Howard) (Howard)]
60   /Names
61     [ (Howard) 10 ]
62 >>
63 endobj
64
65 7 0 obj      % leaf
66 <<
67   /Limits [(Zack) (Zack)]
68   /Names
69     [ (Zack) 70 ]
70 >>
71 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 $\mathbb{R}^m \rightarrow \mathbb{R}^n$ (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 $\mathbb{R}^m \rightarrow \mathbb{R}^n$, 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 $0 \leq i \leq m-1$, Domain _{$2i$} must be less than or equal to Domain _{$2i+1$} . The domain of the function is $\prod_{0 \leq i \leq m-1} [\mathbf{Domain}_{2i}, \mathbf{Domain}_{2i+1}]$
Range	array	(Required for type 0 and 4 functions, otherwise required) an array of $2n$ numbers. Similar to the domain, for every $0 \leq i \leq n-1$, Range _{$2i$} \leq Range _{$2i+1$} . The range (codomain) of the function is $\prod_{0 \leq i \leq n-1} [\mathbf{Range}_{2i}, \mathbf{Range}_{2i+1}]$

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 $[0 \ (\mathbf{Size}_0 - 1) \ 0 \ (\mathbf{Size}_1 - 1) \ \dots]$.
Decode	array	(Optional) an array of $2n$ numbers specifying a linear mapping of sample values into the range 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 $\prod_{0 \leq i < m} \mathbf{Size}_i$ values in the stream. In order, these give the multi-dimensional array

$$g(0, \dots, 0), g(1, \dots, 0), \dots, g(\mathbf{Size}_0 - 1, 0, \dots, 0), g(\mathbf{Size}_0 - 1, 1, \dots, 0), \dots, g(\mathbf{Size}_0 - 1, \dots, \mathbf{Size}_{m-1} - 1)$$

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, \dots, x_{m-1}) , the following steps are taken in order to compute the result:

- (1) Each x_i is clipped to the domain:

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

- (2) The input value is then encoded:

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

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

- (3) Then the input value is clipped

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

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:

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

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

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

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

So for example, suppose we have a sampled function $\mathbb{R}^2 \rightarrow \mathbb{R}$ 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

```
1 14 0 obj
2  <<
3    /FunctionType 0
4    /Domain [-1.0 1.0 -1.0 1.0]
5    /Size [21 31]
6    /Encode [0 20 0 30]
7    /BitsPerSample 4
8    /Range [-1.0 1.0]
9    /Decode [-1.0 1.0]
10   /Length ...
11  >>
12 stream
13 ... 651 sampled values ...
14 endstream
15 endobj
```

1.3.2 Type 2 (Exponential Interpolation) Functions

Type 2 functions are functions $\mathbb{R} \rightarrow \mathbb{R}^n$. 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 $f_0, \dots, f_{k-1}: \mathbb{R} \rightarrow \mathbb{R}^n$ 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 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** = $[f_0, \dots, f_{k-1}]$ each with a domain $[d_{i0}, d_{i1}]$;
- (3) a vector of bounds **Bounds** = $[b_0, \dots, b_{k-2}]$ (and define $b_{-1} = d_0$ and $b_{k-1} = d_1$);
- (4) a vector of encoding values **Encode** = $[e_{00}, e_{01}, \dots, e_{k-1,0}, e_{k-1,1}]$;

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 $f: \mathbb{R} \rightarrow \mathbb{R}^n$, 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:

- Arithmetic 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 $\{\dots\}$. Braces are also used to delimit expressions executed conditionally in **if** and **ifelse** operators.

The **Range** field is required for PostScript functions.

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 T_EX file:

```
hello-world.tex
1 \pdfcompresslevel=0 % don't compress the PDF
2 \nopagenumbers
3 Hello world!
4 \bye
```

we'd get the following PDF:

```
hello-world.pdf
1 %PDF-1.5
2 %
3 3 0 obj
4 <<
5   /Length 66
6 >>
7 stream
8 BT
9   /F1 9.9626 Tf 91.925 759.927 Td [(Hello)-333(w)27(orld!)]TJ
10 ET
11
12 endstream
13 endobj
14 8 0 obj
15 <<
16   /Length1 1472
17   /Length2 9273
18   /Length3 0
19   /Length 10745
20 >>
21 stream
22   %!PS-AdobeFont-1.0: CMR10 003.002
23   %%Title: CMR10
24   %Version: 003.002
25   ...
26 endstream
27 endobj
28 11 0 obj
29 <<
30   /Producer (pdfTeX-1.40.26)
31   ...
32 >>
33 endobj
34 5 0 obj
35 <<
36   /Type /ObjStm
37   /N 7
38   /First 43
39   /Length 1150
40 >>
41 stream
42   2 0 1 106 7 168 9 649 4 878 6 1010 10 1062
43   % 2 0 obj
44   <<
45     /Type /Page
46     /Contents 3 0 R
```

```

47 /Resources 1 0 R
48 /MediaBox [0 0 595.276 841.89]
49 /Parent 6 0 R
50 >>
51 % 1 0 obj
52 <<
53 /Font << /F1 4 0 R >>
54 /ProcSet [ /PDF /Text ]
55 >>
56 % 7 0 obj
57 [ ... ]
58 % 9 0 obj
59 <<
60 /Type /FontDescriptor
61 /FontName /IEQIOR+CMR10
62 /Flags 4
63 /FontBBox [-40 -250 1009 750]
64 /Ascent 694
65 /CapHeight 683
66 /Descent -194
67 /ItalicAngle 0
68 /StemV 69
69 /XHeight 431
70 /CharSet (/H/d/e/exclam/l/o/r/w)
71 /FontFile 8 0 R
72 >>
73 % 4 0 obj
74 <<
75 /Type /Font
76 /Subtype /Type1
77 /BaseFont /IEQIOR+CMR10
78 /FontDescriptor 9 0 R
79 /FirstChar 33
80 /LastChar 119
81 /Widths 7 0 R
82 >>
83 % 6 0 obj
84 <<
85 /Type /Pages
86 /Count 1
87 /Kids [2 0 R]
88 >>
89 % 10 0 obj
90 <<
91 /Type /Catalog
92 /Pages 6 0 R
93 >>
94
95 endstream
96 endobj
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
112 startxref
113 12479
114 %%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:

```

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

```

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

```

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:

```

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 <i>page tree node</i> (see below) that is the root of the document's <i>page tree</i> (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 <i>page label dictionaries</i> (see below).
Names	dictionary	(Optional) The document's name tree (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	integer	(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 <i>content stream</i> (see below) which contains the contents of the page.
Group	dictionary	(Optional) a <i>group attributes dictionary</i> specifying the attributes of the page's page group for use in transparency (see below).
Annots	array	(Optional) an array of <i>annotation dictionaries</i> 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

```

83 % 6 0 obj
84 <<
85 /Type /Pages
86 /Count 1
87 /Kids [2 0 R]
88 >>

```

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

hello-world.pdf

```

43 % 2 0 obj
44 <<
45 /Type /Page
46 /Contents 3 0 R
47 /Resources 1 0 R
48 /MediaBox [0 0 595.276 841.89]
49 /Parent 6 0 R
50 >>

```

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:

```

3 3 0 obj
4 <<
5 /Length 66
6 >>
7 stream
8 BT
9 /F1 9.9626 Tf 91.925 759.927 Td [(Hello)-333(w)27(orld!)]TJ
10 ET
11
12 endstream
13 endobj

```

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

```

51 % 1 0 obj
52 <<
53 /Font << /F1 4 0 R >>
54 /ProcSet [ /PDF /Text ]
55 >>

```

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 The name dictionary

A document may have a *name dictionary*, specified in the **Names** field of the document catalog. This dictionary contains fields which themselves are name trees (see above) which map strings to objects. For example, the name dictionary may have a name tree which maps strings to locations within the document. These names may then be referred to (when specified) in other objects.

The name dictionary may have the following fields:

Key	Type	Value
Dests	name tree	(Optional) a name tree which maps strings to destinations (see below).
AP	name tree	(Optional) a name tree which maps strings to annotation appearance streams (see below).

Other fields exist, but we will not discuss them.

2.2.4 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 coordinate system which is device-independent; it 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 discussed 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

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

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 transpose of the matrix provided here. Nevertheless, the results are the same.

The affine transformation represented by

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

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 <i>clipping path</i> , 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 <i>color space</i> which determines how color values are to be interpreted (e.g. RGB). There are two separate color spaces: one for stroking and one for non-stroking operations. 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 non-stroking operations. 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 <i>blend mode</i> 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 <i>soft-mask</i> 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 non-stroking operations. 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 interpreted 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		<i>Butt cap</i> : the stroke is squared off at the endpoint of the path. There is no projection beyond the end of the path.
1		<i>Round cap</i> : 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		<i>Projecting square cap</i> : 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		<i>Miter join</i> : 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		<i>Round join</i> : an arc of a circle with diameter equal to the line width is drawn around the point where the two segments meet.
2		<i>Bevel join</i> : the 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,...

3.2.4 Graphics state operators

We list the operators which can be used to alter the graphics state. Like all PDF operators, they are postfix.

Operands	Operator	Description
—	q	Push the current graphics state onto the state stack.
—	Q	Pop from the state stack into the current graphics state.

<i>a b c d e f</i>	cm	Modify the current transformation matrix (CTM) by multiplying with the matrix
		$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}.$
<i>lineWidth</i>	w	Sets the line width.
<i>lineCap</i>	J	Sets the line cap style according to the provided code.
<i>lineJoin</i>	j	Sets the line join style according to the provided code.
<i>miterLimit</i>	M	Sets the miter limit.
<i>dashArray dashPhase</i>	d	Set the line dash pattern in the graphics state.
<i>dictName</i>	gs	Set the specified parameters in the graphics state. <i>dictName</i> is the name of a graphics state dictionary in the ExtGState subdictionary of the current Resources dictionary.

As written, you can use the **gs** to alter the graphics state. This is by providing the name of a graphics state parameter dictionary which is provided in the **ExtGState** field of the current **Resources** dictionary. Graphics state parameter dictionaries may have the following fields (all fields are optional):

Key	Type	Value
Type	name	The type of PDF object that this dictionary describes; must be ExtGState .
LW	number	The line width.
LC	integer	The line cap code.
LJ	integer	The line join code.
ML	number	The miter limit.
D	array	The line dash pattern, expressed as an array of the form [<i>dashArray dashPhase</i>] where <i>dashArray</i> is itself an array.
Font	array	An array of the form [<i>font size</i>], where <i>font</i> is an indirect reference to a font dictionary and <i>size</i> is a number expressed in text space units for the font size. These can also be altered by the Tf operator (see below).
BM	name or array	The current blend mode to be used in the transparent imaging model (see below).
SMask	dictionary or name	The current soft mask, which specifies the mask shape or mask opacity values to be used in the transparent imaging model.
CA	number	The current stroking alpha constant, specifying the shape or constant opacity value to be used for stroking operations in the transparent imaging model.
ca	number	Same as CA but for non-stroking operations.
AIS	boolean	The alpha source flag (“alpha is shape”), which specifies if the current soft mask and alpha constant are to be interpreted as shape values (true) or opacity values (false).
TK	boolean	The text knockout flag, which determines the behavior of overlapping glyphs within a text object in the transparent imaging model.

For example, we may have the single-page PDF:

```
1 10 0 obj % the page object
```

```

2  <<
3      /Type /Page
4      /Parent 5 0 R
5      /Resources 20 0 R
6      /Contents 40 0 R
7  >>
8  endobj
9
10 20 0 obj % resource dictionary for the page
11 <<
12     /ProcSet [/PDF /Text]
13     /Font << /F1 25 0 R >>
14     /ExtGState <<
15         /GS1 30 0 R
16         /GS2 35 0 R
17     >>
18 >>
19 endobj
20
21 30 0 obj % first graphics state
22 <<
23     /Type /ExtGState
24     /LW .3
25     /D [[3] 0]
26 >>
27 endobj
28
29 35 0 obj % second graphics state
30 <<
31     /Type /ExtGState
32     /LC 1
33     /LJ 1
34     /LW 1
35 >>
36 endobj
37
38 40 0 obj % contents of the page
39 <<
40     /Length ...
41 >>
42 stream
43 q
44 /GS1 gs
45 0 0 m 10 0 l S
46 /GS2 gs
47 0 -5 m 10 -5 l S
48 Q
49 endstream
50 endobj

```

We have two graphics states, **GS1** and **GS2**, which both alter the graphics state to change how lines are drawn. We now discuss the code for drawing paths.

3.3 Path construction and painting

Paths define the boundaries of areas within the PDF. They can be stroked to draw the boundaries of shapes, or filled, or added to the current clipping path in order to crop material on the page.

A path is made up of one or more disconnected subpaths, each of which is a sequence of connected segments, which are lines or curves. Two segments are connected only if they are defined consecutively: if you define a line from (0,0) to (10,0), then a line to (10,10), the two lines are connected. But if between those two lines you draw another line, they are not connected. This is not entirely true: a subpath may be closed, that is the first and last segments may be connected by the use of a special operator.

There are three types of path operators:

- (1) *path construction operators* which define the geometry of the path;
- (2) *path painting operators* which define how the path should be painted and/or stroked;
- (3) *clipping path operators* which say if a path should be added to the current clipping area.

3.3.1 Path construction operators

Operands	Operator	Description
----------	----------	-------------

$x\ y$	m	Begin a new subpath by moving the current point to coordinate (x, y) in user space.
$x\ y$	l	Append a line from the current point to (x, y) in the current subpath. The new current point is (x, y) .
$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$	c	Append a cubic Bézier curve to the current subpath. The curve extends from the current point to (x_3, y_3) using (x_1, y_1) and (x_2, y_2) as control points.
$x_2\ y_2\ x_3\ y_3$	v	Append a cubic Bézier curve to the current subpath. The curve extends from the current point to (x_3, y_3) using the current point and (x_2, y_2) as control points.
$x_1\ y_1\ x_3\ y_3$	y	Append a cubic Bézier curve to the current subpath. The curve extends from the current point to (x_3, y_3) using (x_1, y_1) and (x_3, y_3) as control points.
—	h	CLOSE the current subpath by appending a straight line segment from the current point to the starting point of the subpath. This terminates the current subpath.
$x\ y\ width\ height$	re	Append a rectangle to the current path as a complete subpath.

Doing

```
1 x y width height re
```

is equivalent to

```
1 x y m
2 (x + width) y l
3 (x + width) (y + height) l
4 x (y + height) l
5 h
```

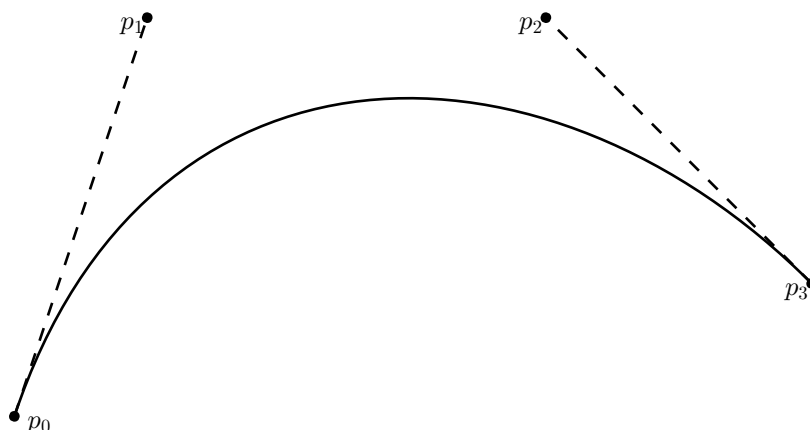
Cubic Bézier curves

Curved path segments are defined using *cubic Bézier curves*. Bézier curves are useful since they can be scaled, and subpaths are also Bézier curves, which makes them very useful in computer graphics.

A cubic Bézier curve is defined by four points: beginning and end points p_0, p_3 , as well as two control points p_1, p_2 . Given these points, the cubic Bézier curve is parameterized by

$$\mathbf{B}(t) = (1 - t)^3 p_0 + 3t(1 - t)^2 p_1 + 3t^2(1 - t) p_2 + t^3 p_3$$

Note that $\mathbf{B}(0) = p_0$ and $\mathbf{B}(1) = p_3$, so this defines a curve between p_0 and p_3 . \mathbf{B} generally does not intersect p_1, p_2 , but it always lies within the convex hull of the four points. Another big reason for the use of Bézier curves is that changing the control points can *intuitively* alter the curvature of the curve.



The above figure demonstrates a useful property of Bézier curves: the lines p_0p_1 and p_2p_3 are tangent to the curve! This helps explain how they can intuitively alter the curvature of the curve.

3.3.2 Path painting operators

We now describe the operators which can be used for painting (stroking and filling) paths:

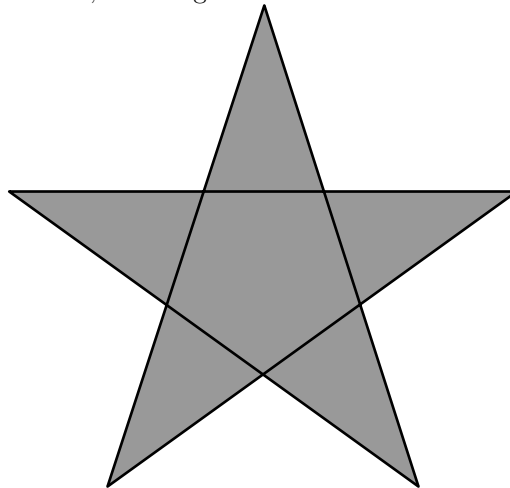
Operands	Operator	Description
—	S	Stroke the path.
—	s	Close and stroke the path. Equivalent to h S .
—	f	Fill the path, using the nonzero winding number rule to determine the region to fill (see below). All open subpaths are implicitly closed.
—	f*	Same as f , except using the even-odd rule.
—	B	Fill then stroke the path, using the nonzero winding number rule.
—	B*	Fill and then stroke the path, using the even-odd rule.
—	b	Close, fill, then stroke the path, using the nonzero winding number rule. (The closing affects only the stroking.)
—	b*	Same as b except using the even-odd rule.
—	n	End the path without stroking or filling it.

The nonzero winding number rule

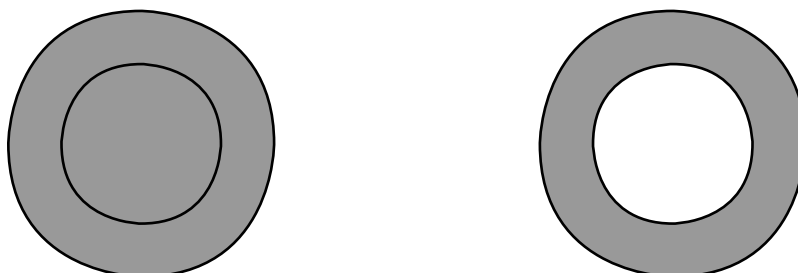
The *nonzero winding number rule* determines whether a given point is inside the path by drawing a ray from the given point to infinity in every direction. For each ray, we count how many times the ray intersects the path. When it intersects the path as it is going left-to-right, it adds one; and when it intersects the path as it is going right-to-left, it subtracts one. If the result is zero (for any ray), the point is outside the path; otherwise it is inside.

Instead of doing this for every ray, an arbitrary ray is chosen.

For convex shapes, this defines the inside and outside as you'd expect. For more complicated boundaries, such as those which intersect themselves, we can get weirder results. For example the five-point star:



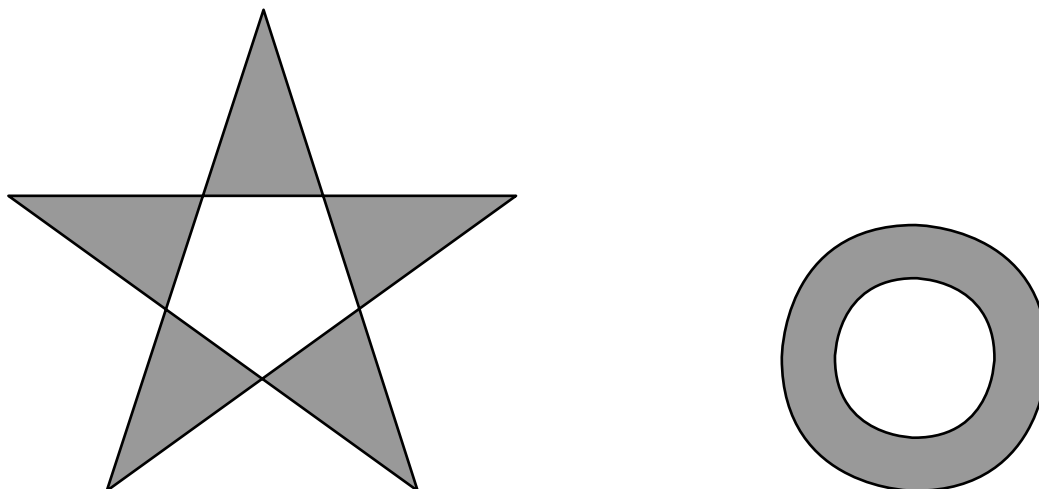
And if we draw a donut shape, the direction we choose to draw it matters. For example, if both circles are drawn in the same direction we get the left, but if they are drawn in opposite directions, we get the right:



The even-odd rule

The even-odd rule is similar to the nonzero winding number rule. Similar to the nonzero winding number rule, a ray is drawn in any direction from the point, and the number of intersections is counted (without any subtractions). If the number is odd, the point is inside; and if it is even, outside.

For convex shapes, this produces the same result as the nonzero winding number rule. But for more complicated shapes, it produces different shapes:



3.3.3 Clipping path operators

The graphics state contains a *current clipping path* which defines a boundary of a shape through which painting operators are cropped. Anything outside the boundary is cropped from the page. The initial current clipping path includes the entire page. After constructing a path and before painting it, a clipping operator (**W** or **W***) may be added. The clipping path is altered to be the intersection of the path with the current clipping path, and this is done only after the path is painted.

Operands	Operator	Description
—	W	Modify the current clipping path by intersecting it with the current path, using the nonzero winding number rule to determine what areas lay inside and outside the path.
—	W*	Like W , except it uses the even-odd rule.

In order to alter the clipping path without painting a path, one can use the **n** operator, which just terminates the path without painting it. So for example, one could do:

```
1 \pdfliteral{q}\rlap{\pdfliteral{0 0 100 4 re W n}}hello\pdfliteral{Q}
```

(`\pdfliteral` injects PDF code into the document, we’ll discuss this later.) This will write “hello”, but only the coordinates which fall inside the rectangle of height 4. This results in:

neuo

It’s important to push the color state before setting the clipping path and pop after you’ve done your clipping. Otherwise all subsequent material will also be cropped.

3.4 Color Spaces

PDF provides the ability to input color in multiple ways. For example, it supports both RGB and CMYK color spaces. The current color space dictates the colors used for stroking and non-stroking operations (there exists separate color spaces for each).

To provide a color, first a color space must be given. We will discuss only the *device color spaces* now, and later we will discuss some *special color spaces*. There are three device color spaces: **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**.

DeviceGray uses only shades of gray, which range between 0.0 and 1.0. **DeviceRGB** uses combinations of red, green, and blue whose values range between 0.0 and 1.0 to produce color. Similarly **DeviceCMYK** uses combination of cyan, magenta, yellow, and black whose values range between 0.0 and 1.0. Importantly, RGB is an additive color system (it mixes light), and CMYK is subtractive (it mixes material; adding colors makes it darker).

We can use the **CS** and **cs** operators to change the stroking and non-stroking color spaces respectively. Their only input is the name of a color space (**DeviceGray**, **DeviceRGB**, **DeviceCMYK**). Then to set the

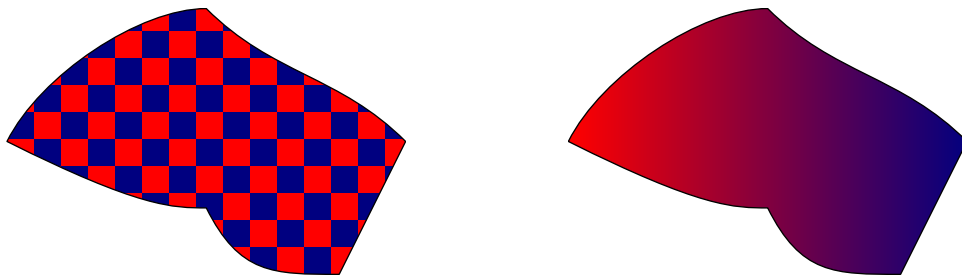
actual color value, one uses **SC** or **sc** to set the color. Their input is a stream of numbers which corresponds to a color value. For example, in the **DeviceRGB** color space the stream must have 3 numbers, while in the **DeviceCMYK** color space it must have 4.

To summarize:

Operands	Operator	Description
<i>name</i>	CS	Sets the current color space for stroking operations. <i>name</i> must be a name object (DeviceGray , DeviceRGB , or DeviceCMYK).
<i>name</i>	cs	Sets the current color space for non-stroking operations. <i>name</i> must be a name object.
$c_1 \dots c_n$	SC	Set the color to be used for stroking operations. Each value c_i must be a number between 0.0 and 1.0. The number of operators n depends on the current color space: 1 for DeviceGray , 3 for DeviceRGB , 4 for DeviceCMYK .
$c_1 \dots c_n$	sc	Same as SC but for non-stroking operations.
<i>gray</i>	G	The same as doing /DeviceGray CS followed by <i>gray</i> SC.
<i>gray</i>	g	The same as doing /DeviceGray cs followed by <i>gray</i> sc.
<i>r g b</i>	RG	The same as doing /DeviceRGB CS followed by <i>r g b</i> SC.
<i>r g b</i>	rg	The same as doing /DeviceRGB cs followed by <i>r g b</i> sc.
<i>c m y k</i>	K	The same as doing /DeviceCMYK CS followed by <i>c m y k</i> SC.
<i>c m y k</i>	k	The same as doing /DeviceCMYK cs followed by <i>c m y k</i> sc.

4 Patterns and Shadings

This section should probably belong to the “Graphics” section; but it’s a big enough of a concept that I (and hopefully you) will let letting it have its own section slide. A pattern is a repeating graphical figure or smooth gradient used to fill regions of space. Take for example the two patterns below:



There are two types of patterns:

- (1) *Tiling patterns* which repeat a small graphical figure (a *pattern cell*) over fixed horizontal and vertical intervals to fill the area to be painted. (The above left example is one of a tiling pattern.)
- (2) *Shading patterns* form a gradient that smoothly transitions between colors and fills the area to be painted. (The above right example is one of a shading pattern.)

Every pattern has a *pattern matrix*, which is a transformation matrix which maps the pattern’s internal coordinate system to the default coordinate system of the pattern’s parent content stream (the content stream from which the pattern was called). The multiplication of the pattern matrix with the transformation matrix of the parent content stream establishes the *pattern coordinate space*. Importantly this is the multiplication of the transformation matrix of the parent stream, and not the CTM. Thus after a pattern is defined in the resources of a content stream, its pattern coordinate space is not affected by changes to the CTM.

4.1 Tiling patterns

A tiling pattern takes a small graphical figure, called the *pattern cell*, and repetitively paints it horizontally and vertically. The pattern cell is a content stream which may contain anything a content stream can contain

(paths, images, XObjects, etc.). When painting using a tiling pattern, the pattern cell is tiled as many times as needed to fill the region specified.

A pattern cell is defined by a content stream, and thus has a stream dictionary. Along with fields common to all stream dictionaries, a pattern dictionary may have the following fields:

Key	Type	Value
Type	name	(Optional) the type of PDF object the dictionary describes; must be Pattern .
PatternType	integer	(Required) a code identifying what kind of pattern this dictionary describes. Must be 1 for a tiling pattern.
PaintType	integer	(Required) a code that determines how the color of the pattern cell is to be specified: <ol style="list-style-type: none"> 1 <i>Colored tiling pattern</i>: the pattern's content stream specifies the colors to be used to paint the pattern cell. When the pattern cell's content stream begins execution, the current color is the one that was in effect in the parent content stream. 2 <i>Uncolored tiling pattern</i>: the pattern's content stream does not specify any color information. Instead, the entire pattern cell is painted a separately specified color each time the pattern is used.
TilingType	integer	(Required) a code that controls adjustments to the spacing of tiles relative to the device pixel grid: <ol style="list-style-type: none"> 1 <i>Constant spacing</i>: pattern cells are spaced consistently, i.e. by a multiple of a device pixel. To achieve this, the application is allowed to distort the pattern cell slightly by making small adjustments to XStep, YStep, and the transformation matrix. This distortion does not exceed one device pixel. 2 <i>No distortion</i>: the pattern cell is not distorted, but the spacing between the pattern cells may vary by as much as one device pixel. 3 <i>Constant spacing and faster tiling</i>: pattern cells are spaced consistently as with <i>constant spacing</i>, but additional distortion is permitted to enable faster tiling.
BBox	array	(Required) an array of four numbers specifying the bottom left and top right corners of the bounding box of the pattern cell. That is, BBox is of the form $[x_0\ y_0\ x_1\ y_1]$ where (x_0, y_0) is the bottom left corner of the pattern cell (relative to its own coordinates), and (x_1, y_1) is the top right corner.
XStep	number	(Required) the desired horizontal spacing between pattern cells, measured in the pattern coordinate system.
YStep	number	(Required) the desired vertical spacing between pattern cells, measured in the pattern coordinate system.
Resources	dictionary	(Required) a resource dictionary containing all the named resources used by the pattern's content stream.
Matrix	array	(Optional) an array of six numbers representing the pattern matrix.

We use the following operators to paint with patterns:

Operands	Operator	Description
----------	----------	-------------

$c_1 \dots c_n$ <i>name</i>	SCN	Sets the current pattern to be stroked, if the current color space is Pattern , or if <i>name</i> refers to a shading pattern. <i>name</i> must refer to a pattern defined in the Pattern subdictionary of the current resource dictionary. If this is an uncolored tiling pattern (PatternType of 1 and PaintType of 2), c_1, \dots, c_n are parameters in the specified underlying color space (see below) which set the color to use for the tiling.
$c_1 \dots c_n$ <i>name</i>	scn	Same as SCN but for non-stroking operations.

For example, to draw the colored tiling pattern at the beginning of the section, the following PDF code was produced:

First we have the pattern object, which defines the pattern cell and necessary data for tiling:

```

1 1 0 obj
2 <<
3   /Type /Pattern      % pattern dictionary
4   /PatternType 1      % tiling pattern
5   /PaintType 1        % colored tiling pattern
6   /TilingType 1       % constant spacing
7   /BBox [0 0 20 20]   % bounding box
8   /XStep 20           % xstep and ystep are precisely
9   /YStep 20           % the size of the pattern cell
10  /Resources << >>    % no resources
11  /Length 84
12 >>
13 stream
14 q
15 1 0 0 rg             % set color to red
16 0 0 10 10 re 10 10 10 10 re f % draw the two red rectangles
17 0 0 .5 rg            % set color to blue
18 10 0 10 10 re 0 10 10 10 re f % draw the two blue rectangles
19 Q
20 endstream
21 endobj

```

Then we have the content stream (which here is an XObject) in which the pattern is painted.

```

1 2 0 obj
2 <<
3   /Type /XObject      % fields for the XObject...
4   /Subtype /Form
5   /BBox [0 0 149.44 99.626]
6   /FormType 1
7   /Matrix [1 0 0 1 0 0]
8   /Resources 9 0 R     % resources are in object 9 0
9   /Length 193
10 >>
11 strea
12 q
13 .996264 0 0 .996264 0 0 cm % this will be explained later
14 .5 0 0 .5 0 0 cm         % scale down by half
15 1 j 1 J                 % rounded lines
16 /Pattern cs             % set current non-stroking color space to be a pattern
17 /P scn                 % set the current non-stroking pattern
18 0 100 m                % draw the region to paint...
19 25 150 100 200 150 200 c
20 200 150 250 150 300 100 c
21 275 50 250 0 y
22 200 0 175 0 150 50 c
23 125 50 100 50 0 100 c
24 h
25 B                      % fill
26 Q
27 endstream
28 endobj

```

The resources for this XObject are in object 9 0. This object must define the name /P. Indeed it does:

```

1 % 9 0 obj
2 <<
3   /Pattern << /P 1 0 R >>

```

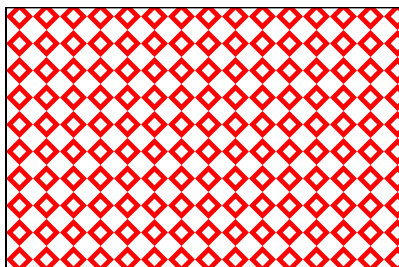
```

4  /ProcSet [ /PDF ]
5  >>

```

So /P is the pattern defined in object 1 0.

For an uncolored tiling pattern, we must also specify what color we want to paint it with. So when we call **CS** or **cs**, we need to tell PDF that we're switching to a **Pattern** color space, as well as the underlying color space (e.g. **DeviceRGB**). Then when we call **SCN** or **scn**, we need to tell PDF what color we want to paint the pattern with. For example, suppose we have the following uncolored pattern:



The pattern cell here is simply a hollowed-out diamond. The PDF code for the pattern cell is as follows:

```

1  1 0 obj
2  <<
3      /Type /Pattern          % pattern cell
4      /PatternType 1          % tiling pattern
5      /PaintType 2            % uncolored tiling pattern
6      /TilingType 1           % constant spacing
7      /BBox [0 0 10 10]      % bounding box
8      /XStep 10               % xstep and ystep are precisely
9      /YStep 10               % the size of the pattern cell
10     /Resources << >>        % no resources
11     /Length 66
12  >>
13  stream
14      5 10 m
15      0 5 1 5 0 1 10 5 1 h    % draw outer diamond
16      5 7.5 m
17      2.5 5 1 5 2.5 1 7.5 5 1 h % draw inner diamond
18      f*                      % fill using even-odd
19                              % (so that the diamond is hollowed out)
20  endstream
21  endobj

```

The code which draws the pattern is as follows:

```

1  2 0 obj
2  <<
3      /Type /XObject
4      /Subtype /Form
5      /BBox [0 0 149.44 99.626]
6      /FormType 1
7      /Matrix [1 0 0 1 0 0]
8      /Resources 9 0 R
9      /Length 78
10  >>
11  stream
12      q
13      .996264 0 0 .996264 0 0 cm    % to be explained later
14      1 j 1 J                      % rounded lines
15      /CSP cs                      % set color space to /CSP
16      1 0 0 /P scn                 % set pattern to /P with color 1 0 0
17      0 0 150 100 re              % draw rectangle
18      B                          % and fill
19      Q
20  endstream
21  endobj

```

We see that here we set the (non-stroking) color space to a name **CSP**. This name is defined in the resource dictionary, in object 9 0:

```

1 % 9 0 obj
2 <<
3   /ColorSpace << /CSP [ /Pattern /DeviceRGB ] >>
4   /Pattern << /P 1 0 R >>
5   /ProcSet [ /PDF ]
6 >>

```

Here **CSP** is defined to be a **ColorSpace**. But instead of being defined as a name like **DeviceRGB**, it is defined to be an array of two names: **Pattern** and **DeviceRGB**. This is because since **P** is an uncolored tiling pattern, we must specify the color to color it with. In object 2 0, we specified the color code 1 0 0, but in order to interpret this color code we need to know what the underlying color space is. That's why the second color space in the array is **DeviceRGB**, telling us to set the underlying color space to be **DeviceRGB**.

4.2 Shading Patterns

A shading pattern is one which defines a gradient between colors. It is a pattern for all intents and purposes, and painting it is done in the same way as a normal pattern. Unlike a tiling pattern, a shading pattern is defined by a dictionary instead of a content stream. Fields in the pattern dictionary for a shading pattern are as follows:

Key	Type	Value
Type	name	(Optional) the type of PDF object the dictionary describes; must be Pattern .
PatternType	integer	(Required) the code defining the type of pattern this dictionary describes. Must be 2 for a shading pattern.
Shading	dictionary or stream	(Required) a shading object (see below) defining the shading pattern gradient.
Matrix	array	(Optional) An array of six numbers defining the pattern matrix.
ExtGState	dictionary	(Optional) a graphics state parameter dictionary to be put into effect when the shading is painted.

The most important field is **Shading**, which defines the shading object; the object which controls the look of the gradient. A shading object may be a content stream or a dictionary, and in either case we refer to the dictionary (either the object itself or the content stream's dictionary) as the shading dictionary. The shading dictionary may have the following fields, which are common to all types of shadings (but different types of shadings will have additional fields):

Key	Type	Value
ShadingType	integer	(Required) the shading type (discussed below): <ol style="list-style-type: none"> 1 function-based shading; 2 axial shading; 3 radial shading; 4 free-form Gourand-shaded triangle mesh; 5 lattice-form Gourand-shaded triangle mesh; 6 coons patch mesg; 7 tensor-product patch mesh.
ColorSpace	name	(Required) the color space in which the color values are to be interpreted. This cannot be a Pattern space.
Background	array	(Optional) an array containing a color value (appropriate to ColorSpace) determining a background color value. If it is present, this is used to color any region outside of the region defined by the shading object.
BBox	array	(Optional) an array of the form $[x_0 \ y_0 \ x_1 \ y_1]$ which defines the bottom-left and top-right coordinates of the bounding box to clip the shading to.

We will focus on shading types 1 through 3, as 4 through 7 are much more complicated. If you're interested, consult the PDF reference.

The term *target coordinate space* means the pattern coordinate space, unless the shading is invoked by the shading operator **sh**, in which case it refers to the current user space. Recall that the pattern coordinate space is *not* affected by changes to the CTM.

4.2.1 Type 1 (Function-Based) Shadings

A function-based shading is one where the color at every point on the gradient is defined by a specified function. This is the most general form of shading. If a shading object is specified to be of type 1, it may have the following fields:

Key	Type	Value
Domain	array	(Optional) an array of four numbers $[x_0 \ x_1 \ y_0 \ y_1]$ specifying the rectangular domain of coordinates over which the color function(s) are defined. Default is $[0 \ 1 \ 0 \ 1]$.
Matrix	array	(Optional) an array of six numbers specifying a transformation mapping the coordinate space specified by Domain to the shading's target coordinate space. For example, to map the default Domain to a one-inch square starting at coordinate (100,100), the Matrix would be $[72 \ 0 \ 0 \ 72 \ 100 \ 100]$ (since the default value of UserUnit is 1/72th of an inch).
Function	function	A 2-in, n -out function; or an array of n 2-in 1-out functions (where n is the number of color values required by the ColorSpace specified in the shading dictionary). Each function's domain must be a superset of Domain .

For example, suppose we want to shade using the following function, which simply computes the average of its two inputs:

```

1 1 0 obj
2 <<
3   /FunctionType 4
4   /Domain [0 1 0 1]
5   /Range [0 1]
6   /Length 13
7 >>
8 stream
9 {
10    add
11    2 div
12 }
13 endstream
14 endobj

```

We create a shading object which uses this function:

```

1 % 2 0 obj
2 <<
3   /ShadingType 1           % functional shading
4   /ColorSpace /DeviceGray  % use grayscale coloring
5   /Domain [0 1 0 1]        % domain is [0,1] x [0,1]
6   /Matrix [100 0 0 100 0 0] % scale gradient by 100
7   /Function 1 0 R          % use above function
8 >>

```

and a shading pattern object which sets this as its **Shading** attribute

```

1 % 4 0 obj
2 <<
3   /Type /Pattern % pattern object
4   /PatternType 2 % shading pattern
5   /Shading 2 0 R % shading object is above dictionary
6 >>

```

The object in which the pattern is drawn is

```

1 5 0 obj
2 <<
3   /Type /XObject
4   /Subtype /Form
5   /BBox [0 0 99.626 99.626]
6   /FormType 1
7   /Matrix [1 0 0 1 0 0]
8   /Resources 11 0 R
9   /Length 42
10 >>
11 stream
12 q
13 /Pattern cs      % set color space to pattern
14 /P scn           % set current pattern
15 0 0 100 100 re   % draw rectangle
16 B               % fill
17 Q
18 endstream
19 endobj

```

Notice that we don't need to change the color space, as the color space is determined by the shading pattern (**P** is defined in the resources of this object, object 11 0 as a reference to object 4 0). The resulting pattern is:



4.2.2 Type 2 (Axial) Shadings

Axial shadings are shadings which vary along a linear axis between two endpoints, and extends indefinitely perpendicular to that axis. The shading may also extend indefinitely beyond either or both endpoints by continuing the boundary colors. The following table lists the fields specific to type 2 shadings:

Key	Type	Value
Coords	array	(Required) an array of the form $[x_0 \ y_0 \ x_1 \ y_1]$ which specifies the start and endpoints of the axis, expressed in the shading's target coordinate space.
Domain	array	(Optional) an array of two numbers $[t_0 \ t_1]$ which specifies the limiting values of the parameteric value t (i.e. $t_0 \leq t \leq t_1$). Default value is $[0 \ 1]$.
Function	function	(Required) a 1-in n -out function, or an array of n 1-in 1-out functions (where n is the number of color values required by the Color Space). These functions are used to determine the color along the axis. Each function's domain must be a superset of Domain .
Extend	array	(Optional) an array of two boolean values which specifies whether to extend the shading beyond the start and end points of the axis, respectively. Default value is $[\text{false} \ \text{false}]$.

Suppose we want to create a linear gradient between red and blue. We can use the following linear interpolation function

```

1 % 1 0 obj
2 <<
3   /FunctionType 2 % exponential interpolation function
4   /Domain [0 1]   % domain is [0,1]
5   /C0 [1 0 0]     % interpolate between red ...
6   /C1 [0 0 .5]     % ... and blue

```

```

7      /N 1          % using linear interpolation
8  >>

```

Now we define the shading object

```

1  % 3 0 obj
2  <<
3      /ShadingType 2          % axial shading
4      /ColorSpace /DeviceRGB % use RGB color sapce
5      /Coords [0 0 100 0]    % axis between origin and (100,0)
6      /Domain [0 1]          % domain of function is [0 1]
7      /Function 1 0 R         % function is above linear interpolation function
8      /Extend [false false]  % do not extend
9  >>

```

and the pattern object

```

1  % 4 0 obj
2  <<
3      /Type /Pattern % pattern object
4      /PatternType 2 % shading pattern
5      /Shading 3 0 R % using the above shading object
6  >>

```

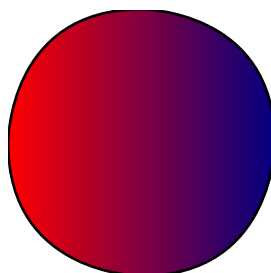
And the actual content stream where the gradient is drawn is

```

1  5 0 obj
2  <<
3      /Type /XObject
4      /Subtype /Form
5      /BBox [0 0 99.626 99.626]
6      /FormType 1
7      /Matrix [1 0 0 1 0 0]
8      /Resources 11 0 R
9      /Length 94
10 >>
11 stream
12 q
13 /Pattern cs
14 /P scn
15 50 100 m
16 100 100 100 50 v 100 0 50 0 v 0 0 0 50 v 0 100 50 100 v % draw circle
17 h
18 B                                     % fill
19 Q
20 endstream
21 endobj

```

Again, **P** is defined in the **Resources** of the content stream as a reference to the pattern object defined. The resulting pattern is



4.2.3 Type 3 (Radial) Shadings

Radial shadings define a gradient that varies between two circles. This can be used to depict three-dimensional spheres and cones. Radial shading dictionaries may contain the following fields:

Key	Type	Value
Coords	array	(Required) an array of the form $[x_0 \ y_0 \ r_0 \ x_1 \ y_1 \ r_1]$, which specifies the centers and radii of the starting and ending circles. r_0 and r_1 must be at least zero. If one is zero, it is treated as a point, and if both are, nothing is painted.

Domain	array	(Optional) an array of two numbers $[t_0 \ t_1]$ specifying the limiting values of the parametric value t . This variable varies linearly between t_0 and t_1 as it goes between the two circles. t is the input to the function specified by Function .
Function	function	(Required) a 1-in n -out, or an array of n 1-in 1-out functions, (where n is the number of values in the ColorSpace) which specify the color to be drawn, called with parametric value t . Each function's domain must be a superset of Domain .
Extend	array	(Optional) an array of two boolean values which specifies whether or not to extend the shading beyond the starting and ending circles. Default value <i>[false false]</i> .

For example, if we want to shade a circle with an inner color of red and an outer color of blue, we first define the interpolation function which linearly interpolates between red and blue (same as our previous example). Then we define a shading object:

```

1 % 3 0 obj
2 <<
3   /ShadingType 3           % radial shading
4   /ColorSpace /DeviceRGB   % rgb color space
5   /Coords [75 75 0 50 50 50] % first circle is a point at (75,75),
6                               % second is the circle of radius 50 at (50,50)
7   /Domain [0 1]           % domain of linear interpolation function
8   /Function 1 0 R         % use linear interpolation function
9   /Extend [false false]   % do not extend
10 >>

```

The object which draws the pattern is

```

1 4 0 obj
2 <<
3   /Type /XObject
4   /Subtype /Form
5   /BBox [0 0 99.626 99.626]
6   /FormType 1
7   /Matrix [1 0 0 1 0 0]
8   /Resources << 10 0 R
9   /Length 7
10 >>
11 stream
12 /Sh sh
13 endstream
14 endobj

```

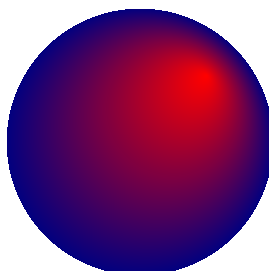
Notice here we just use the **sh** operator, which we explain below. Let's look at the resources dictionary:

```

1 % 10 0 obj
2 <<
3   /Shading << /Sh 3 0 R >>
4   /ProcSet [ /PDF ]
5 >>

```

So we define **Sh** to be a **Shading** object, referencing the shading object defined in object 3 0. The resulting pattern is



4.2.4 The shading operator

If you have a shading pattern *name* and you'd like to draw it as-is (without using it to fill a path), you can use the shading operator **sh**. *name* must be specified in the **Shading** subdictionary of the resource dictionary of the current content stream. This operator should only be applied to bounded or geometrically defined shadings.

The operand to the shading operator **sh** must be a *shading object*; not a *pattern object*.

5 External Objects

An external object (XObject) is a graphics object whose content is defined by a self-contained content stream, separate from the content stream in which it is used. There are three types of XObjects, but we won't discuss the third (PostScript XObjects) as their use is discouraged. The other two types of XObjects are

- (1) *image XObjects* which represent images;
- (2) *form XObjects* which are self-contained descriptions of an arbitrary sequence of graphics objects. Two subtypes of form XObjects are *group XObjects* and *reference XObjects*.

If an XObject is named by *name* in the **XObject** subdictionary of the **Resource** dictionary of a content stream, it can be painted using the **Do** operator. For example, we may define the following form XObject:

```
1 1 0 obj
2 <<
3   /Type /XObject
4   /Subtype /Form
5   /BBox [0 0 10 1]
6   /FormType 1
7   /Matrix [1 0 0 1 0 0]
8   /Resources 7 0 R
9   /Length ...
10 >>
11 stream
12 1 w
13 1 0 0 1 0 .5 cm
14 0 0 m 10 0 l S
15 endstream
16 endobj
```

which simply paints a line of length 10. Now, in another content stream we can paint this XObject using the **Do** operator:

```
1 4 0 obj
2 <<
3   /Length 34
4 >>
5 stream
6 q
7 1 0 0 1 72 765.905 cm
8 /Fm1 Do
9 Q
10 endstream
11 endobj
```

We see that **Do** references the name **Fm1**, which is defined in the page's **Resource** dictionary to be a reference to object 1 0.

```
1 % 2 0 obj
2 <<
3   /XObject << /Fm1 1 0 R >>
4   /ProcSet [ /PDF ]
5 >>
```

5.1 Images

There are two ways to insert images into a PDF. One is using image XObjects, and another is using inline images where the content of the image is represented inline directly in a content stream.

5.1.1 Image XObjects

An external image file may be read into a PDF and referenced via an image XObject. An image XObject is a content stream: the stream itself represents the image data, and the dictionary (the image dictionary) contains the image's metadata.

An image is two-dimensional array of samples, each sample representing a color. Each sample consists of as many color components as are needed for the color space in which they are specified (1 for **DeviceGray**, 3 for **DeviceRGB**, 4 for **DeviceCMYK**). Each component may be a 1-, 2-, 4-, 8-, 16- bit integer.

Fields in the image dictionary may include:

Key	Type	Value
Type	name	(Optional) the type of PDF object this dictionary describes; must be XObject .
Subtype	name	(Required) the type of XObject this dictionary describes; must be Image .
Width	integer	(Required) the width of the image, in samples.
Height	integer	(Required) the height of the image, in samples.
ColorSpace	name	(Required for images; not permitted for image masks (see below)) the color space in which the color samples reside.
BitsPerComponent	integer	(Required except for image masks) the number of bits used to represent each color component. This may be 1, 2, 4, 8, or 16.
ImageMask	boolean	(Optional) a flag which indicates whether this image shall be an image mask (see below). If true , BitsPerComponent must be 1, and ColorSpace and Mask must be left unspecified. Default: false .
Mask	stream or array	(Optional; not permitted for image masks) an image XObject defining an image mask to apply to this image; or an array specifying a range of colors to be applied as a color key mask (see below). If ImageMask is true , must be left undefined.
Decode	array	(Optional) an array of numbers describing how to map image samples into the range of values appropriate for the image's color space. If ImageMask is true , must be either $[0\ 1]$ or $[1\ 0]$. Otherwise it must be an array of length $2n$ where n is the number of color components in the color space. The default value is $[0\ 1\ \dots\ 0\ 1]$.
SMask	stream	(Optional) an image XObject which defines a soft-mask image for the current image. The alpha value is determined by the luminosity of each pixel, and affects the shape or opacity depending on the AIS flag in the current graphics state.

Decode decodes the samples so that the color components fall into the range required for the color space. Given a sample of $c_1 \dots c_n$ and a decode array of $[D_{11}\ D_{12} \dots D_{n1}\ D_{n2}]$, the color components are decoded to be

$$c'_i = \text{Interpolate}(c_i; 0, 2^b - 1, D_{i1}, D_{i2})$$

where b is the **BitsPerComponent**.

Masked images

Generally, in the opaque image model (see below on transparency), an image takes a rectangular area on the page. Every pixel in the rectangular area is painted according to the image. But in PDF, an image may specify a **Mask**, which is itself an image which determines which samples are drawn (unmasked) and which are not (masked) or an array determining which colors in the image to show. If **Mask** specifies an image, it must be an *image mask*; an image whose **ImageMask** is **true**.

An image mask is an image where every sample is a bit (**BitsPerComponent** is 1), and **ColorSpace** must be left unspecified (since the samples don't represent colors). A "black" bit is unmasked; the underlying image will "bleed" through, changing the color of the area of the sample. A "white" bit is masked; the underlying image will not change the color of the area of the sample.

The **Decode** array of an image mask must be either $[0\ 1]$; in which case 0 is a "black" bit and 1 a "white" bit. Or it may be $[1\ 0]$ in which case 0 is "white" and 1 is "black".

An image mask may not have the same **Width** and **Height** as the image it is masking. This is okay, since images are painted in a 1×1 square, so their areas will line up on the page regardless.

Color key masking

Instead of an image mask being specified for **Mask**, instead an array of $2n$ integers may be specified, where n is the number of color components in the **ColorSpace** of the image. Let this array be $[m_1\ M_1 \dots m_n\ M_n]$ where $m_i, M_i \in [0, 2^b - 1]$ (b is **BitsPerComponent**). A sample of value $c_1 \dots c_n$ (before being decoded)

is masked (not painted) if all of its components lie in $[m_i, M_i]$, i.e. it is not painted if $m_i \leq c_i \leq M_i$ for all $i \in [1, n]$.

For example, we may have the following image:

```

1 1 0 obj
2 <<
3   /Type /XObject
4   /Subtype /Image
5   /Width 9
6   /Height 9
7   /Filter /ASCIIHexDecode
8   /ColorSpace /DeviceRGB
9   /BitsPerComponent 8
10  /Mask [255 255 255 255 255 255]
11  /Length 570
12 >>
13 stream
14   ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000
15   ff0000 ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
16   ff0000 ffffffff 00ff00 00ff00 ffffffff 00ff00 00ff00 ffffffff ff0000
17   ff0000 ffffffff 00ff00 00ff00 ffffffff 00ff00 00ff00 ffffffff ff0000
18   ff0000 ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
19   ff0000 ffffffff 0000ff ffffffff ffffffff ffffffff 0000ff ffffffff ff0000
20   ff0000 ffffffff ffffffff 0000ff 0000ff 0000ff ffffffff ffffffff ff0000
21   ff0000 ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
22   ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 >
23 endstream
24 endobj

```

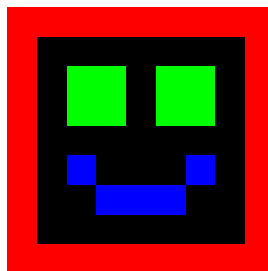
The image is painted with all white pixels masked out. We can then paint it in the following XObject, with a black background:

```

1 2 0 obj
2 <<
3   /Type /XObject
4   /Subtype /Form
5   /BBox [0 0 100 100]
6   /FormType 1
7   /Matrix [1 0 0 1 0 0]
8   /Resources 9 0 R
9   /Length 52
10 >>
11 stream
12 0 0 100 100 re f
13 q 100 0 0 100 0 0 cm /SIm Do Q
14 endstream
15 endobj

```

This results in



5.1.2 Inline images

As opposed to an image XObject, which is defined as an external (and reusable) object, an inline image may be defined entirely within the content stream in which it is used. An inline image is declared in a content stream in the following way:

```

1 BI % begin inline image object
2 ... key-value pairs ...
3 ID % begin image data
4 ... image data ...
5 EI % end inline image object

```

That is, the **BI** operator begins the inline image dictionary (except its just a stream of key-value pairs, not wrapped in `<<...>>`), **ID** ends it and begins the image data (which specifies how to color each pixel), and **EI**

ends the inline image object. The fields that can be placed in between **BI** and **ID** are a subset of the fields which may appear in an image XObject dictionary:

Key	Type	Value
BitsPerComponent	integer	(Required) the number of bits used to represent each color component (a single value in a color value). Valid values are 1, 2, 4, 8, 16.
Width	integer	(Required) the width of the image, in samples.
Height	integer	(Required) the height of the image, in samples.
ColorSpace	name	(Required) the color space that the image is defined in.
Filter	array	(Required) an array of filters to be applied to the image data to decode it (like any stream).

We can use the following abbreviations for the fields (only in inline images, not image XObjects):

- **BitsPerComponent** — **BPC**
- **Width** — **W**
- **Height** — **H**
- **ColorSpace** — **CS**
- **Filter** — **F**

So if **CS** requires n values per color (e.g. 3 for RGB), **W** is w , and **H** is h , the image data should have $n \cdot w \cdot h$ samples. Each sample must have a size of **BPC** after being decoded by the filter. The only filter we will discuss is **AHx** (or **ASCIIHexDecode**), which decodes ASCII hex codes. That is, it takes two ASCII characters (which are either 0 – 9 or $a - z$ or $A - Z$), and computes the hex value. Whitespace is ignored.

The stream must be ended by a > character, as seen below.

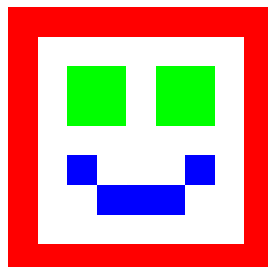
For example, we may have the inline image:

```

1 q
2 .996264 0 0 .996264 0 0 cm
3 100 0 0 100 0 0 cm
4 BI
5   /BPC 8
6   /CS /DeviceRGB
7   /F [ /AHx ]
8   /W 9
9   /H 9
10 ID
11   ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000
12   ff0000 ffffff ffffff ffffff ffffff ffffff ffffff ffffff ff0000
13   ff0000 ffffff 00ff00 00ff00 ffffff 00ff00 00ff00 ffffff ff0000
14   ff0000 ffffff 00ff00 00ff00 ffffff 00ff00 00ff00 ffffff ff0000
15   ff0000 ffffff ffffff ffffff ffffff ffffff ffffff ffffff ff0000
16   ff0000 ffffff 0000ff ffffff ffffff ffffff 0000ff ffffff ff0000
17   ff0000 ffffff ffffff 0000ff 0000ff 0000ff ffffff ffffff ff0000
18   ff0000 ffffff ffffff ffffff ffffff ffffff ffffff ffffff ff0000
19   ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 ff0000 >
20 EI
21 Q

```

Which draws the following:



Recall that an image is by default one unit by one unit, so we must scale it to get the desired size.

5.2 Form XObjects

A form XObject is a self-contained content stream which can be painted within other content streams. The graphics defined in a form XObject are defined relative to its *form coordinate space*, which can then be scaled to user space using its **Matrix** field.

Form XObjects may have the following fields in their stream dictionary, as well as fields common to all content streams:

Key	Type	Value
Type	name	(Optional) the type of PDF object the dictionary describes; must be XObject .
Subtype	name	(Required) the type of XObject the dictionary describes; must be Form .
BBox	array	(Required) a bounding box which defines the boundary of the XObject in form space. The XObject is clipped to this boundary.
Matrix	array	(Optional) a matrix which represents the transformation from form to user space.
Resources	dictionary	(Required) a dictionary specifying the resources required by the form XObject.
Group	dictionary	(Optional) a <i>group attributes dictionary</i> indicating that the contents of the form XObject are to be treated as a group and specifying the attributes of that group (see below).

5.2.1 Group XObjects

A *group XObject* is a type of form XObject that can be used to group graphical elements together for various purposes. It is declared by adding the **Group** field to a form XObject's dictionary. This is a dictionary called the *group attributes dictionary*, which may have the following fields:

Key	Type	Value
Type	name	(Optional) the type of PDF object that this dictionary describes; must be Group .
S	name	(Required) the <i>group subtype</i> , which identifies the kind of group the dictionary describes. The only available group subtype is Transparency . Transparency group XObjects are discussed later.

6 Transparency

6.1 The idea

6.1.1 Compositing semi-transparent images

In a normal, opaque, image model, every pixel is given a color in some color space like RGB. When two objects are placed on top of one another, the bottom object has no effect on the color of the pixels which overlap with the top object. In a transparent image model, objects may be given an opacity value (called an α -value) which determines how transparent the object is. Thus, if a semi-transparent object (an object with an α -value less than 1) is placed on top of another object, the bottom object's color will affect the pixels that overlap.

We use the following notation: the underlying (opaque) color space is denoted by \mathcal{C} . Colors in the color space \mathcal{C} are denoted with an uppercase C and some subscript to uniquely distinguish them. A color in the transparent color space whose underlying space is \mathcal{C} is denoted with a lowercase c ; it is made up of an opaque color $C \in \mathcal{C}$ and an α -value $\alpha \in [0, 1]$ where $\alpha = 0$ means the color is totally transparent, and an α -value of 1 means the color is totally opaque.

How should we interpret the α value of a color (the α -component of an image is also called the α channel)? The idea that we will use is that given any pixel (or area of an image), if it is colored with color C and α -value α , then the color C is uniformly spread out over the pixel to take up an α fraction of the space. For example,

if we want to color a pixel with RGB value $(1, 0, 0)$ and $\alpha = 0.5$, then half the pixel will be colored red $(1, 0, 0)$, so the resulting color will be $(0.5, 0, 0)$.

In order to store a transparent color (C, α) , it is often useful to instead store the tuple $(c = \alpha C, \alpha)$. This is since most procedures around transparent colors use αC instead of α ; and multiplying every component by α in every pixel is time-consuming.

Now suppose we'd like to blend two pixels A and B . The first has color (C_A, α_A) and the second has color (C_B, α_B) . We split the pixel into four regions: the region with neither A nor B (which doesn't contribute to the pixel color, so we ignore it); the region with A and not B (denoted $A - B$); the region with B and not A (denoted $B - A$); and the region with both A and B (denoted $A \cap B$). Each of these regions takes up a certain amount of area in the pixel, and thus has an associated α -value. We can compute the α -values using the assumption that colors are spread uniformly. So if a region has an α -value of α_1 , and another has an α -value of α_2 , their overlap has an α -value of $\alpha_1 \cdot \alpha_2$ (due to uniformity). Thus

- region $A - B$ has an α -value of $\alpha_A(1 - \alpha_B)$ (since the region outside B , B^c , has an α -value of $1 - \alpha_B$);
- similarly region $B - A$ has an α -value of $\alpha_B(1 - \alpha_A)$;
- region $A \cap B$ has an α -value of $\alpha_A \cdot \alpha_B$.

Now we have to determine which color each region has. Obviously $A - B$ and $B - A$ should have colors C_A and C_B respectively. But what about $A \cap B$? For this reason we must use a *blending function* $\mathfrak{B}: \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$, which takes two opaque colors and outputs how they should be blended. We leave this as a user-defined variable. So $A \cap B$ will have a color of $\mathfrak{B}(C_A, C_B)$.

Thus the transparent color of the resulting pixel $A \cup B$ as a whole will be

$$\alpha_{A-B}C_{A-B} + \alpha_{B-A}C_{B-A} + \alpha_{A \cap B}C_{A \cap B} = \alpha_A(1 - \alpha_B)C_A + \alpha_B(1 - \alpha_A)C_B + \alpha_A\alpha_B\mathfrak{B}(C_A, C_B)$$

That is,

$$c_{A \cup B} = \alpha_A(1 - \alpha_B)C_A + \alpha_B(1 - \alpha_A)C_B + \alpha_A\alpha_B\mathfrak{B}(C_A, C_B)$$

This is called the *basic color compositing formula*. Recall that $c_{A \cup B} = \alpha_{A \cup B} \cdot C_{A \cup B}$, so now we can ask ourselves: what is the value of $\alpha_{A \cup B}$ and the value of the underlying opaque color $C_{A \cup B}$?

Well we can write the region $A \cup B$ as a disjoint union $(A - B) \sqcup B$, thus

$$\alpha_{A \cup B} = \alpha_{A-B} + \alpha_B = \alpha_A(1 - \alpha_B) + \alpha_B = \alpha_A + \alpha_B - \alpha_A\alpha_B$$

Then, we get the following formula for $C_{A \cup B}$:

$$C_{A \cup B} = \left(1 - \frac{\alpha_B}{\alpha_{A \cup B}}\right) \cdot C_A + \frac{\alpha_B}{\alpha_{A \cup B}} \cdot [(1 - \alpha_A) \cdot C_B + \alpha_A \cdot \mathfrak{B}(C_A, C_B)]$$

The benefit of this formula, over the naive one obtained by simply dividing by $\alpha_{A \cup B}$, is that only one division needs to be performed.

Note that $A \cup B$ is not the best choice of notation for this region: it is not commutative. That is, $C_{A \cup B}$ does not necessarily equal $C_{B \cup A}$. This is because the blend function may not be commutative. Instead, we will use the notation B/A , or B over A .

To summarize, the α -premultiplied color of B over A is

$$c_{B/A} = \alpha_A(1 - \alpha_B)C_A + \alpha_B(1 - \alpha_A)C_B + \alpha_A\alpha_B\mathfrak{B}(C_A, C_B)$$

and the color of B over A is

$$C_{B/A} = \left(1 - \frac{\alpha_B}{\alpha_{B/A}}\right) \cdot C_A + \frac{\alpha_B}{\alpha_{B/A}} \cdot [(1 - \alpha_A) \cdot C_B + \alpha_A \cdot \mathfrak{B}(C_A, C_B)]$$

We will define this to be $C_{B/A} = \text{BASICCOMPOSITE}(C_A, \alpha_A, C_B, \alpha_B, \mathfrak{B})$. and the α -value of B over A is

$$\alpha_{B/A} = \alpha_{A-B} + \alpha_B = \alpha_A(1 - \alpha_B) + \alpha_B = \alpha_A + \alpha_B - \alpha_A\alpha_B$$

It is useful to define the union function $\cup: [0, 1]^2 \rightarrow [0, 1]$ by $x \cup y = 1 - (1 - x)(1 - y) = x + y - xy$. Thus, $\alpha_{B/A} = \alpha_A \cup \alpha_B$.

This gives us the theoretical background for composing two semi-transparent images together. We now discuss different types of blend functions:

6.1.2 Blend functions

A blend function is a function $\mathfrak{B}: \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$. Generally color spaces \mathcal{C} are sets of tuples, these are the only kind of color space we will consider. In fact, we will only consider color spaces of the form $[0, 1]^n$ for some n .

A blend function \mathfrak{B} is called *separable* there exists a function $[0, 1] \times [0, 1] \rightarrow [0, 1]$ (which we will also denote \mathfrak{B}), such that

$$\mathfrak{B}((c_b^1, \dots, c_b^n), (c_s^1, \dots, c_s^n)) = (\mathfrak{B}(c_b^1, c_s^1), \dots, \mathfrak{B}(c_b^n, c_s^n))$$

That is, the blend of two colors (the backdrop C_b and the new source C_s) is the blend of each component.

The separable blend functions supported by PDF are as follows:

Name	Result
Normal	Selects the source color, ignoring background: $\mathfrak{B}(c_b, c_s) = c_s$
Multiply	Multiplies the backdrop with the source: $\mathfrak{B}(c_b, c_s) = c_b \cdot c_s$ This can also be thought of as taking the “intersection” of the colors. The result is always at most as light as the darkest color.
Screen	Takes the complement of the multiplication of the complements: $\mathfrak{B}(c_b, c_s) = 1 - [(1 - c_b) \cdot (1 - c_s)] = c_b + c_s + c_b c_s$ This can be thought of as taking the “union” of the colors. The result is always at most as dark as the lightest color.
HardLight	Multiplies or screens the colors (intersection or union), depending on the source color value. The effect is similar to shining a harsh spotlight on the backdrop: $\mathfrak{B}(c_b, c_s) = \begin{cases} \text{Multiply}(c_b, 2c_s) & c_s \leq 0.5 \\ \text{Screen}(c_b, 2c_s - 1) & c_s > 0.5 \end{cases}$
Overlay	Multiplies or screens the colors (intersection or union), depending on the backdrop color value. $\mathfrak{B}(c_b, c_s) = \text{HardLight}(c_s, c_b)$
Darken	Selects the darker color $\mathfrak{B}(c_b, c_s) = \min(c_b, c_s)$
Lighten	Selects the lighter color $\mathfrak{B}(c_b, c_s) = \max(c_b, c_s)$
ColorDodge	Brightens the backdrop color to reflect the source color: $\mathfrak{B}(c_b, c_s) = \begin{cases} \min(1, c_b / (1 - c_s)) & c_s < 1 \\ 1 & c_s = 1 \end{cases}$

ColorBurn Darkens the backdrop color to reflect the source color:

$$\mathfrak{B}(c_b, c_s) = \begin{cases} 1 - \min(1, (1 - c_b)/c_s) & c_s > 0 \\ 0 & c_s = 0 \end{cases}$$

SoftLight Darkens or lightens the colors, depending on source color value. The effect is similar to shining a diffused spotlight on the backdrop:

$$\mathfrak{B}(c_b, c_s) = \begin{cases} c_b - (1 - 2c_s)c_b(1 - c_b) & c_s \leq 0.5 \\ c_b + (2c_s - 1)(D(c_b) - c_b) & c_s > 0.5 \end{cases}$$

where

$$D(x) = \begin{cases} x((16x - 12)x + 4) & x \leq 0.25 \\ \sqrt{x} & x > 0.25 \end{cases}$$

Difference Subtracts the darker from the lighter color

$$\mathfrak{B}(c_b, c_s) = |c_b - c_s|$$

Exclusion Produces an effect similar to **Difference** but with less contrast.

$$\mathfrak{B}(c_b, c_s) = c_b + c_s - 2c_b c_s$$

PDF also supports non-separable blend functions. These blend functions follow essentially the same principal:

- (1) convert both colors from the blending space to an intermediate HSL (hue-saturation-luminosity) representation;
- (2) create a new color from some combination of the HSL values of the colors;
- (3) transform the color back to the blending color space.

The non-separable functions use the following auxillary functions (we assume the blending space is RGB; for a color C , we denote it by (C_r, C_g, C_b) ; we also let C_{min} be the minimum of the color components, C_{mid} the middle, and C_{max} the maximum):

1. **function** LUM(C)
2. **return** $0.3C_r + 0.59C_g + 0.11C_b$
3. **end function**
4. **function** SETLUM(C, ℓ)
5. $\Delta = \ell - Lum(C)$
6. **return** $ClipColor(C + (\Delta, \Delta, \Delta))$
7. **end function**
8. **function** CLIPCOLOR(C)
9. $\ell = Lum(C)$
10. **if** ($C_{min} < 0$)
11. $C_r = \ell + (C_r - \ell) \cdot \ell / (\ell - C_{min})$
12. $C_g = \ell + (C_g - \ell) \cdot \ell / (\ell - C_{min})$
13. $C_b = \ell + (C_b - \ell) \cdot \ell / (\ell - C_{min})$
14. **end if**
15. **if** ($C_{max} > 1$)
16. $C_r = \ell + (C_r - \ell) \cdot (1 - \ell) / (C_{max} - \ell)$
17. $C_g = \ell + (C_g - \ell) \cdot (1 - \ell) / (C_{max} - \ell)$
18. $C_b = \ell + (C_b - \ell) \cdot (1 - \ell) / (C_{max} - \ell)$
19. **end if**
20. **return** C
21. **end function**
22. **function** SAT(C)
23. **return** $C_{max} - C_{min}$

24. **end function**

```

25. function SETSAT( $C, s$ )
26.   if ( $C_{max} > C_{min}$ )
27.      $C_{mid} = (C_{mid} - C_{min}) \cdot s / (C_{max} - C_{min})$ 
28.      $C_{max} = s$ 
29.   else
30.      $C_{mid} = 0$ 
31.      $C_{max} = 0$ 
32.   end if
33.    $C_{min} = 0$ 
34.   return  $C$ 
35. end function

```

The non-separable blend functions supported by PDF are:

Name	Result
Hue	Creates a color with the hue of the source and saturation and luminosity of the backdrop: $\mathfrak{B}(C_b, C_s) = \text{SETLUM}(\text{SETSAT}(C_s, \text{SAT}(C_b)), \text{LUM}(C_b))$
Saturation	Creates a color with the saturation of the source color and the hue and luminosity of the backdrop. $\mathfrak{B}(C_b, C_s) = \text{SETLUM}(\text{SETSAT}(C_b, \text{SAT}(C_s)), \text{LUM}(C_b))$
Color	Creates a color with the luminosity of the backdrop and hue and saturation of the source: $\mathfrak{B}(C_b, C_s) = \text{SETLUM}(C_s, \text{LUM}(C_b))$
Luminosity	Creates a color with the luminosity of the source and hue and saturation of the backdrop: $\mathfrak{B}(C_b, C_s) = \text{SETLUM}(C_b, \text{LUM}(C_s))$

6.1.3 α -values and shape and opacity

The α -value of an object is actually dictated by two other parameters: *shape* and *opacity*. These are denoted by f and q respectively, and they range between 0 and 1. The α -value of a pixel is actually the product of its shape and opacity: $\alpha = f \cdot q$. When the shape of an object at a pixel is 0, its opacity is undefined there. We adopt the convention that $0/0 = 0$.

Shape and opacity can be derived from multiple different sources:

- Objects can provide an *object shape* and *object opacity*. Elementary objects such as strokes, fills, and text have an intrinsic shape. The value of this shape is 1 for points inside the object and 0 for those outside. The shape of a group object is the union of the shapes of the objects it contains. We denote object shape by f_j .

Elementary objects have an object opacity, denoted q_j , of 1 everywhere.

- A *soft mask* is a source of shape and opacity independent of other objects. It can be used to alter the shape and opacity of another object, for example if a soft mask specifies a gradient which slowly goes from opaque to transparent, applying this to text will have the effect of making the text fade out. The shape of a soft mask is denoted f_m , and its opacity is q_m .
- The shape and opacity can be altered by a scalar constant. The shape constant is denoted f_k and the opacity constant is denoted q_k .

The shape and opacity of a shape at a point are called the *source shape* and *source opacity*, which are defined to be

$$f_s = f_j \cdot f_m \cdot f_k, \quad q_s = q_j \cdot q_m \cdot q_k$$

Then the *source- α -value* is defined to be $f_s \cdot q_s$.

Compositing two objects together must form a result shape and opacity. If the source shape and opacity are f_s, q_s respectively, and the backdrop shape and opacity are f_r, q_r then we know the following:

$$\alpha_r = \alpha_s \cup \alpha_b, \quad f_r = f_s \cup f_r$$

thus we get that $q_r = \alpha_r / f_r = (\alpha_s \cup \alpha_b) / (f_s \cup f_r)$.

6.1.4 Transparency groups

In a PDF, objects form what is called a *transparency stack*, which determines how the objects are layered in the transparent imaging model. Suppose the transparency stack looks like E_1, \dots, E_n , then we begin with some *initial backdrop*, then composite element E_1 onto it, then E_2 on that, and so on.

Each element in the transparency stack may itself be a group of elements, called a *transparency group*; and the elements in a transparency group may be transparency groups as well, and so on. So the transparency stack is more like a tree structure. We can view the transparency stack as a whole as a single transparency group.

Each element in a transparency group is an object specifying the element's color, shape, opacity, and blend mode.

For a given transparency group G , we consider three backdrops:

- (1) the *group backdrop* is the result of compositing all elements up to, but not including, the first element in the group (for non-knockout groups, discussed later).
- (2) the *initial backdrop* is a backdrop that is selected for compositing the group's first element against. This is either the group backdrop (for non-isolated groups) or a transparent backdrop (for isolated groups, see below).
- (3) the *immediate backdrop* of an element in the group E_i is the result of compositing all elements in the group up to but not including it.

To composite a group G , we start with a backdrop which specifies a color C_0 and an α -value α_0 . Then the function $\text{COMPOSITE}(C_0, \alpha_0, G)$ returns a triple (C, f, α) which is the group's color, the group's shape, and the group's α -value. Once the group is composited, it is treated like a new single item on the outer transparency stack.

Suppose a non-isolated non-knockout group G is composed of E_1, \dots, E_n . We define $\text{COMPOSITE}(C_0, \alpha_0, G)$ recursively as follows:

1. **function** $\text{COMPOSITE}(C_0, \alpha_0, G)$
2. $f_{g_0} = \alpha_{g_0} = 0$
3. **for** $(i = 1, \dots, n)$
 - ▷ C_{s_i} is the source color of element i , f_{j_i} is the object shape of element i ,
and $\alpha_{j_i} = f_{j_i} \cdot q_{j_i}$ is the object α -value of element i .
 4. **if** (E_i is a group)
 5. $(C_{s_i}, f_{j_i}, \alpha_{j_i}) = \text{COMPOSITE}(C_{i-1}, \alpha_{i-1}, E_i)$
 6. **else if** (E_i is an object)
 7. C_{s_i} is the intrinsic color of the object
 8. f_{j_i} is the intrinsic shape of the object
 9. α_{j_i} is the intrinsic α -value of the object
 10. **end if**
 - ▷ f_{s_i} is the source shape of element i .
It is obtained from the product of the object shape with the mask shape and shape constant.
 11. $f_{s_i} = f_{j_i} \cdot f_{m_i} \cdot f_{k_i}$
 - ▷ α_{s_i} is the source α -value of element i .
It is obtained from the product of the object α , with the mask α and α constant.
 12. $\alpha_{s_i} = \alpha_{j_i} \cdot (f_{m_i} \cdot q_{m_i}) \cdot (f_{k_i} \cdot q_{k_i})$
 - ▷ f_{g_i} is the accumulated shape of group elements E_1, \dots, E_i excluding the backdrop.
 13. $f_{g_i} = f_{g_{i-1}} \cup f_{s_i}$
 - ▷ α_{g_i} is the accumulated source α of group elements E_1, \dots, E_i excluding the backdrop.
 14. $\alpha_{g_i} = \alpha_{g_{i-1}} \cup \alpha_{s_i}$
 - ▷ α_i is the accumulated α after compositing E_i onto the rest of the group, including the backdrop.
 15. $\alpha_i = \alpha_0 \cup \alpha_{g_i}$
 - ▷ C_i is the accumulated color after compositing E_i .
 16. $C_i = \left(1 - \frac{\alpha_{s_i}}{\alpha_i}\right) \cdot C_{i-1} + \frac{\alpha_{s_i}}{\alpha_i} \cdot [(1 - \alpha_{i-1}) \cdot C_{s_i} + \alpha_{i-1} \cdot \mathfrak{B}_i(C_{i-1}, C_{s_i})]$
 17. **end for**

```

18.    $C = C_n + (C_n - C_0) \cdot \left( \frac{\alpha_0}{\alpha_{g_n}} - \alpha_0 \right)$ 
19.    $f = f_{g_n}$ 
20.    $\alpha = \alpha_{g_n}$ 
21.   return  $(C, f, \alpha)$ 
22. end function

```

Let's take a look at each variable. α_{s_i} is defined as you'd expect: it's precisely $\alpha_{j_i} \cdot \alpha_{m_i} \cdot \alpha_{k_i}$. If we look at the definition of α_i , we see that it's equal to

$$\alpha_i = \alpha_0 \cup \alpha_{g_i} = \alpha_0 \cup \alpha_{s_i} \cup \alpha_{g_{i-1}} = \dots = \alpha_0 \cup \bigcup_{j \leq i} \alpha_{s_j}$$

Thus, we have that

$$\alpha_i = \alpha_{i-1} \cup \alpha_{s_i}$$

This means that line 16 is another way of writing

$$C_i = \text{BASICCOMPOSITE}(C_{i-1}, \alpha_{i-1}, C_{s_i}, \alpha_{s_i}, \mathfrak{B}_i)$$

What's the rationale behind the α_i variable? Well, α_i is the α of the group composited with its backdrop. We don't want to store this in α_{g_i} , as it shouldn't affect later computations of compositing elements with the immediate backdrop. Importantly, every element of a group is composited onto a backdrop including the initial backdrop. This is done because most blend modes are dependent on the backdrop and source colors. This is also precisely the difference between non-isolated and isolated transparency groups.

The formulas at the end (after the loop) essentially remove the contribution of the initial backdrop from the computation. This is since when the group is composited with the initial backdrop, it may be composited with a different blend mode or mask. This is more clear by viewing it as follows: first we define the *backdrop fraction* which measures the relative contribution of the backdrop color to the overall color:

$$\phi_b = \frac{(1 - \alpha_{g_n})\alpha_0}{\alpha_0 \cup \alpha_{g_n}}$$

Indeed, $(1 - \alpha_{g_n})\alpha_0$ measures the area covered by the backdrop and not the group. And $\alpha_0 \cup \alpha_{g_n}$ is the total area covered. So ϕ_b measures the ratio between the area covered by just the backdrop and the total area. Then we get

$$C = \frac{C_n - \phi_b \cdot C_0}{1 - \phi_b}$$

Notice that $C_n - \phi_b \cdot C_0$ has the effect of removing the backdrop color from C_n , since ϕ_b is the amount that C_0 contributed. And

$$\frac{1}{1 - \phi_b} = \frac{\alpha_0 \cup \alpha_{g_n}}{\alpha_{g_n}}$$

so we are looking at the inverse ratio between the amount of area the group takes and the total area. So multiplying $C_n - \phi_b \cdot C_0$ by this scales back up the color to take the entirety of the area.

Isolated groups

An isolated group differs from a non-isolated groups in that each element is composited with a transparent backdrop as opposed to the group's backdrop. The effect that this has is that the group itself is transparent *relative to itself*, but not to its backdrop unless transparency is specified. This is generally what you want.

The only change to the above algorithm is that α_0 is set to zero.

Knockout groups

In a knockout group, each element is composited with the group's initial backdrop rather than the stack of preceding elements in the group. If elements have a binary shape (1 inside, 0 outside), this has the effect of overlaying (or knocking-out) the effect of elements beneath it. This is similar to the opaque imaging model, except that "topmost object wins" applies to both color and opacity.

We alter the algorithm above as follows:

- α_{g_i} is instead defined to be $(1 - f_{s_i})\alpha_{g_{i-1}} + \alpha_{s_i}$
- we also define

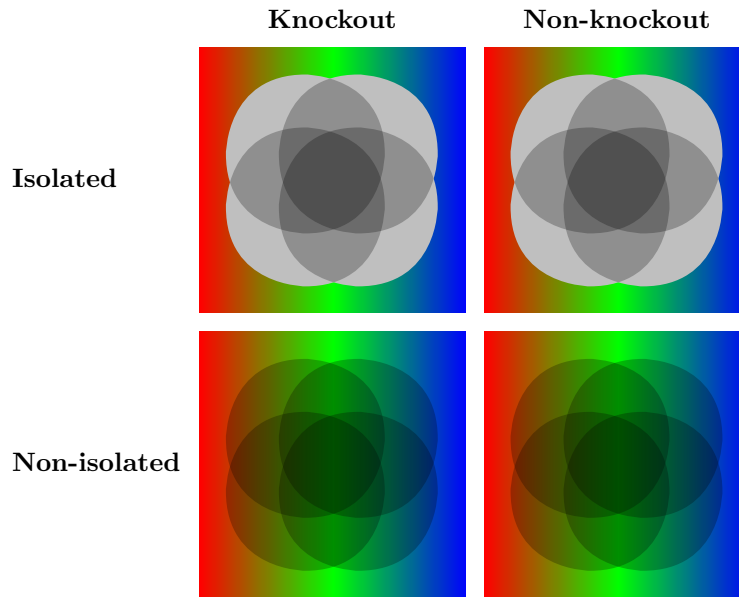
$$C_t = (f_{s_i} - \alpha_{s_i})\alpha_b C_b + \alpha_{s_i} \cdot [(1 - \alpha_b)C_{s_i} + \alpha_b \mathfrak{B}_i(C_b, C_{s_i})]$$

and then define

$$C_i = \frac{(1 - f_{s_i})\alpha_{i-1}C_{i-1} + C_t}{\alpha_i}$$

Differences between types of transparency groups

The following table shows the differences between types of transparency groups. The backdrop of each image is a gradient. The transparency group consists of four overlapping circles, painted with opacity of 1 (in the group). The group is then painted with opacity of 1 onto the backdrop. The blend mode in the group is **Multiply**, and the blend mode onto the backdrop is **Normal**.



The result for isolated groups is as you'd expect: the backdrop has no effect on the group since the opacity is 1 (opaque). The overlapping areas are colored darker due to the **Multiply** blend mode. For non-isolated groups, the backdrop does affect the group, even though it is opaque.

6.1.5 Soft masks

As previously discussed, shape and opacity values can be altered by soft masks, which are just other sources of shape and opacity values. These are generalizations of clipping paths, as a soft mask which specifies binary values does the same as clipping.

A mask can be defined by creating a transparency group and painting objects into it, which defines the color, shape, and opacity in the usual way. Then the resulting group can be used to derive the mask in one of two ways, discussed below.

A soft mask specifies a single value, called the *mask value* at every point. The mask value either specifies shape or opacity according to a flag (alpha is shape) in the graphics state.

Deriving a soft mask from group alpha

The first method involves first computing the color, shape, and α -value of a transparency group G using the normal formula

$$(C, f, \alpha) = \text{COMPOSITE}(C_0, \alpha_0, G)$$

where C_0, α_0 are arbitrary and don't contribute to the result. Since C isn't used, we don't need to compute it.

The user then specifies a *transfer function* $[0, 1] \rightarrow [0, 1]$ which maps α to a mask value.

Deriving a soft mask from group luminosity

The second method of deriving a soft mask from a transparency group utilizes the luminosity of the group. The group is composited with a fully opaque backdrop of some color, then the mask value is defined to be the luminosity of the resulting color, C . The color C is created by first compositing G with a fully opaque background C_0 , this gives

$$(C_g, f_g, \alpha_g) = \text{COMPOSITE}(C_0, 1, G)$$

then C is the weighted average between C_0 and C_g :

$$C = (1 - \alpha_g) \cdot C_0 + \alpha_g \cdot C_g$$

The color is then converted to its luminosity as follows:

- if the underlying color space is **DeviceGray**, the luminosity is the same as the value;
- if the underlying color space is **DeviceRGB**, the luminosity is

$$0.3 \cdot C_r + 0.59 \cdot C_g + 0.11 \cdot C_b$$

- if the underlying color space is **DeviceCMYK**, the luminosity is

$$0.3 \cdot (1 - C_c)(1 - C_k) + 0.59 \cdot (1 - C_m)(1 - C_k) + 0.11 \cdot (1 - C_y)(1 - C_k)$$

Then the value computed is passed to a transfer function $[0, 1] \rightarrow [0, 1]$ to compute the mask value.

6.2 Specifying transparency in PDF

Single graphic objects are treated as elementary graphics objects in a transparency stack. Even if they overlap themselves (like a path which intersects itself), this does not affect its transparency. An object's source color C_s is the same as its color in the opaque imaging model. Its backdrop color C_b is the result of previous painting operations.

The current blend mode is specified in the current graphics state under the **BM** entry. The **BM** entry is either a name corresponding to one of the names given above, or an array of names. In the latter case, the first name that device recognizes is used (or **Normal** if it recognizes none of them). This way more blending modes can be implemented in the future.

The current soft mask can be specified in the current graphics state under the **SMask** entry. The value is a *soft mask dictionary*, defined below:

6.2.1 Soft mask dictionaries

A soft mask dictionary may have the following fields:

Key	Type	Value
Type	name	(Optional) the type of PDF object this dictionary describes; must be Mask .
S	name	(Required) a subtype specifying the method to be used to derive the mask values from the transparency group specified by G . This is either <ul style="list-style-type: none"> • Alpha: compute the group's α-value and pass it to the transfer function specified by TR to derive the mask value. • Luminosity: derive the mask value from the group's computed luminosity.
G	stream	(Required) a transparency group XObject (see below) to be used as the source of the α or luminosity values for the mask. If S is Luminosity , the transparency group XObject must contain a CS entry defining the color space in which the compositing is to be performed.
BC	array	(Optional) an array of component values specifying the backdrop to be used as the initial backdrop when compositing G . This is only used if S is Luminosity . The array must have n values, where n is the number of components in the XObject specified by G 's CS entry. Default value: whatever the code is for black in CS .
TR	function or name	(Optional) a 1-in 1-out function specifying the transfer function to be used in deriving the mask values from α or luminosity values. The name Identity may be used in place of a function to represent the identity function. Default value: Identity .

6.2.2 Transparency group XObjects

A transparency group is represented in PDF as a special subtype of group XObject, called a *transparency group XObject*. A group XObject is in turn a subtype of a form XObject, distinguished by the presence of a **Group** entry in its form dictionary, whose value is a *group attributes dictionary*. For a transparency group, this may have the following fields:

Key	Type	Value
-----	------	-------

S	name	(Required) the <i>group subtype</i> , specifying what kind of group this dictionary describes. Must be Transparency .
CS	name or array	(Required) the group color space, which is used for the following: <ol style="list-style-type: none"> (1) as the color space into which colors are converted when painted into the group; (2) as the blending color space in which objects are composited within the group; (3) as the color space of the group as a whole when it in turn is painted as an object onto its backdrop.
I	boolean	(Optional) a flag specifying if this transparency group is isolated. Default value: false .
K	boolean	(Optional) a flag specifying if this transparency group is a knockout group. Default value: false

6.3 An example

Suppose we want to create the following effect of fading in the word “hello”:

hello

The idea is as follows: we create a soft mask containing the word “hello”, and then in another XObject we draw the gradient and use the soft mask as a mask. The code to do this is as follows:

```

1 11 0 obj                                % XObject which draws the gradient and sets the soft mask
2 <<
3 /Type /XObject
4 /Subtype /Form
5 /BBox [0 0 20.479 6.918]
6 /FormType 1
7 /Matrix [1 0 0 1 0 0]
8 /Resources 16 0 R
9 /Length 48
10 >>
11 stream
12 q
13 .996264 0 0 .996264 0 0 cm
14 /GS1 gs                                % set soft mask
15 /Sh sh                                  % draw gradient
16 Q
17 endstream
18 endobj
19
20 % 16 0 obj                                resources of above XObject
21 <<
22   /Shading << /Sh 3 0 R >>
23   /ExtGState << /GS1 10 0 R >>
24   /ProcSet [ /PDF ]
25 >>
26
27 % 3 0 obj                                shading object
28 <<
29   /ShadingType 2
30   /ColorSpace /DeviceGray
31   /Coords [0 6.94444 0 0]               % 6.94 is height of "hello"
32   /Domain [0 1]
33   /Function 1 0 R
34   /Extend [false false]
35 >>
36
37 % 1 0 obj                                linear interpolation between white and black
38 <<
39   /FunctionType 2
40   /Domain [0 1]
41   /C0 [1]
42   /C1 [0]
43 /N 1 >>
44
45 % 10 0 obj                                graphics state object

```

```

46 <<
47     /Type /ExtGState
48     /SMask 9 0 R
49 >>
50
51 % 9 0 obj                mask object
52 <<
53     /Type /Mask
54     /S /Luminosity
55     /G 6 0 R
56 >>
57
58 6 0 obj                  % transparency group XObject containing "hello"
59 <<
60     /Type /XObject
61     /Subtype /Form
62     /Group 5 0 R
63     /BBox [0 0 20.479 6.918]
64     /FormType 1
65     /Matrix [1 0 0 1 0 0]
66     /Resources 7 0 R
67     /Length 71
68 >>
69 stream
70 1 g 1 G
71 1 0 0 1 0 6.918 cm
72 BT
73 /F1 9.9626 Tf 0 -6.918 Td [(hello)]TJ
74 ET
75 endstream
76 endobj
77
78 % 5 0 obj                group attributes dictionary
79 <<
80     /Type /Group
81     /S /Transparency
82     /CS /DeviceGray
83 >>

```

7 Text

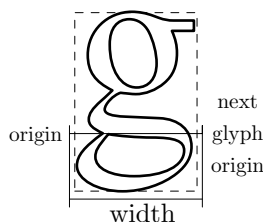
We begin by distinguishing between two concepts: *characters* and *glyphs*. Characters are abstract concepts (like the character “A”), while a glyph is a specific rendering of a character (e.g. *A*, **A**, *Ⓐ* are all glyphs of the character “A”). A *font* defines the glyphs for a specific character set: for example computer modern roman defines the glyphs for a set of standard Latin characters. Fonts are defined in a form of program, called a *font program*, usually stored in a separate file, and written in a specialized language such as the *Type 1* or *TrueType* font formats. PDFs utilize a special kind of dictionary, called a *font dictionary*, to identify the font program and additional information about the font.

A content stream can paint glyphs on a page by first specifying a font dictionary and then a string object which is interpreted as a sequence of character codes specifying glyphs in the font. The *glyph description*, stored in the font program, provides a sequence of graphics operators to execute in order to paint the glyph. A PDF consumer renders glyphs by then executing this sequence (for performance, glyphs can be cached).

7.1 Glyph structure

A glyph has a *glyph coordinate system*, which is the coordinate system in which it is defined. All graphics operations related to painting a glyph are done relative to the glyph coordinate system. Except for Type 3 fonts, units in the glyph space are 1/1000th of a user space unit.

A glyph has the following structure:



The *bounding box* is the dashed box around the glyph. It is the smallest box which encompasses the glyph shape. The *origin* of the glyph is point (0,0) in the glyph coordinate system. The *glyph displacement* is a vector from

the origin which gives the coordinate where the next glyph's origin should be placed. A glyph displacement that is horizontal is generally called the *width*. We deal here with only horizontal writing modes.

7.2 Text state

The current *text state* is composed of nine parameters which affect only the painting of text. These are:

Parameter	Description
T_c	Character spacing
T_w	Word spacing
T_h	Horizontal scaling
T_l	Leading
T_f	Text font
T_{fs}	Text font size
T_{mode}	Text rendering mode
T_{rise}	Text rise
T_k	Text knockout

T_f and T_{fs} are self-explanatory. We will not discuss T_k . Within a text object, two further parameters are defined: T_m the text matrix; T_{lm} the text line matrix; and T_{rm} the text rendering matrix.

The parameters can be altered using the following operators:

Operands	Operator	Description
<i>charSpace</i>	Tc	Sets the character space T_c to <i>charSpace</i> which is a unit in unscaled text space units. This is used by Tj , TJ , and ' operators. Initial value: 0.
<i>wordSpace</i>	Tw	Sets the word space T_w to <i>wordSpace</i> which is a unit in unscaled text space units. This is used by Tj , TJ , and ' operators. Initial value: 0.
<i>scale</i>	Tz	Set the horizontal scaling T_h to <i>scale</i> /100. <i>scale</i> is a number specifying the percentage of the normal width. Initial value: 100.
<i>leading</i>	TL	Set the text leading T_l to <i>leading</i> , a number expressed in unscaled text space units. Text leading is used by T* , ', and " operators.
<i>font size</i>	Tf	Set the text font T_f to <i>font</i> and font size T_{fs} to <i>size</i> . <i>font</i> is the name of a font resource in the Font subdictionary of the current resource dictionary. There is no initial value for either of these parameters; they must be specified before painting glyphs.
<i>render</i>	Tr	Set the text rendering mode T_{mode} to <i>render</i> , an integer. Initial value: 0.
<i>rise</i>	Ts	Set the text rise T_{rise} to <i>rise</i> , which is a number in unscaled text space units. Initial value: 0.

A parameter in *unscaled* text space units means it is specified in a coordinate system defined by the text matrix T_m , but not scaled by the font size T_{fs} .

7.2.1 Character spacing

The character spacing parameter T_c specifies the amount of spacing between characters. It is specified in unscaled text space, but is subject to scaling by the horizontal scaling parameter T_h . T_c is added to the horizontal component of the widths displacement. For example:

$T_c = 0$: Character

$T_c = 2$: Character

7.2.2 Word spacing

Word spacing acts similarly to character spacing, but only affects the space character, code 32. \TeX treats spaces differently, and thus this parameter doesn't have an effect on strings written using \TeX .

7.2.3 Horizontal scaling

T_h specifies how much the horizontal component of the glyphs should be stretched as a percentage. This also affects T_c (character spacing) and T_w (word spacing).

$T_h = 100$: Word

$T_h = 50$: WordWord

7.2.4 Leading

T_l specifies the distance between baselines (like $\text{\backslash baselineskip}$). This is not used by \TeX .

7.2.5 Text rendering mode

The text rendering mode T_{mode} determines how glyphs are shown. It can cause glyphs to be stroked, filled, or added to the clipping path (or some combination of the three). The following table describes the text rendering modes:

Mode	Example	Description
0	R	Fill text
1	R	Stroke text
2	R	Fill, then stroke text
3		Neither fill nor stroke text (invisible)
4	R	Fill text and add to path for clipping
5	R	Stroke text and add to path for clipping
6	R	Fill, then stroke text, and add to path for clipping
7	R	Add to path for clipping

The text rendering mode must not change inside a text object (see below). The glyphs are accumulated inside a text object, and at the end form a path where the outlines of each glyph are considered as subpaths. Then text, if specified, is filled using the nonzero winding number rule. The current clipping path, if specified, is intersected with this path (after all the stroking and filling operations). This remains in effect until an invocation of the \Q operator.

7.2.6 Text rise

Text rise, T_{rise} , specifies the distance in unscaled text space units, to move the baseline up or down from its default location. For example:

```
1 (This ) Tj
2 -5 Ts
3 (text ) Tj
4 5 Ts
5 (moves ) Tj
6 0 Ts
7 (around) Tj
```

will produce

This_{text} moves around

7.3 Text objects

A PDF text object consists of operators that can show text strings, move the text position, and set text state and other parameters. Additionally, three parameters are defined only within a text object and do not persist between text objects:

- (1) the *text matrix*, T_m ;
- (2) the *text line matrix*, T_{lm} ;
- (3) the *text rendering matrix*, T_{rm} . This is actually just an intermediate result that combines the effects of text state parameters, the text matrix, and the CTM.

A text object has the form

```
1 BT
2 ... text or other allowed operators ...
3 ET
```

BT begins a text object, which initializes T_m and T_{lm} to the identity matrix. **ET** ends the text object. Text objects cannot be nested.

7.3.1 Text space details

Text is shown in *text space*, which is defined by a combination of the text matrix T_m , and the text state parameters T_{fs} (font size), T_h (horizontal scaling), and T_{rise} (text rise). This determines how text coordinates are transformed into user space. The transformation from text space to user space is represented by an intermediate object, the *text rendering matrix*, T_{rm} , which is:

$$T_{rm} = CTM \cdot T_m \cdot \begin{pmatrix} T_{fs} \cdot T_h & 0 & 0 \\ 0 & T_{fs} & T_{rise} \\ 0 & 0 & 1 \end{pmatrix}$$

That is, first we map a point (x, y) according to the text state parameters, to

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} T_{fs} \cdot T_h \cdot x \\ T_{fs} \cdot y + T_{rise} \end{pmatrix}$$

then we map this according to the text matrix and the CTM to user space. T_{rm} is an intermediate result, ostensibly recomputed before each glyph is painted.

After a glyph is painted, the text matrix is updated according to the glyph's displacement and any spacing parameters which may apply. We define

$$t_x = \left(\left(w - \frac{T_j}{1000} \right) \cdot T_{fs} + T_c + T_w \right) \cdot T_h$$

where

- w is the width of the glyph;
- T_j is a positioning adjustment (specified in a **TJ** array, see below);
- T_{fs}, T_h are text state parameters corresponding to font size and horizontal scaling, respectively.
- T_c, T_w are text state parameters corresponding to the current character- and word-spacing parameters in the graphics state (only applied if necessary; e.g. T_w is only applied between words).

In addition to T_m , another parameter is stored: T_{lm} . This is the value of T_m at the beginning of the line. This makes it easier to move to the next line: simply set T_m to T_{lm} displaced by some displacement. In particular, if you want to move to the next line, displaced by (Δ_x, Δ_y) , set

$$T_m = T_{lm} = T_{lm} \cdot \begin{pmatrix} 1 & 0 & \Delta_x \\ 0 & 1 & \Delta_y \\ 0 & 0 & 1 \end{pmatrix}$$

7.3.2 Text placement operators

The following operators alter T_m and T_{lm} :

Operands	Operator	Description
----------	----------	-------------

$\Delta_x \Delta_y$	Td	Move to the start of the next line, offset from the start of the current line by $\Delta = (\Delta_x, \Delta_y)$. Δ_x, Δ_y are expressed in unscaled text space units. This has the effect of doing $T_m = T_{lm} = T_{lm} \cdot \begin{pmatrix} 1 & 0 & \Delta_x \\ 0 & 1 & \Delta_y \\ 0 & 0 & 1 \end{pmatrix}$
$\Delta_x \Delta_y$	TD	Move to the start of the next line, offset from the start of the current line by $\Delta = (\Delta_x, \Delta_y)$. As a side effect, sets the leading parameter in the text state. This has the same effect as doing $-\Delta_y$ TL and then $\Delta_x \Delta_y$ Td .
$a b c d e f$	Tm	Sets the text matrix and text line matrix $T_m = T_{lm} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$
—	T*	The operands are all numbers. The matrices are not composed with the previous values of T_m or T_{lm} . Move to the start of the next line. This is the same as doing 0 T_l Td , where T_l is the current leading parameter in the text state.

7.3.3 Text-showing operators

The following operators show text on the page. They reposition the glyphs as they do so.

Operands	Operator	Description
<i>string</i>	Tj	Show a text string.
<i>string</i>	'	Move to the next line and show a text string. This has the same effect as doing T* and then <i>string</i> Tj .
$a_w a_c$ <i>string</i>	"	Move to the next line and show a text string, using a_w for word spacing and a_c for character spacing. This has the same effect as doing a_w Tw , a_c TC , and then <i>string</i> ' .
<i>array</i>	TJ	Show one or more text strings, allowing individual glyph positioning. Each element of <i>array</i> can be a string or a number. If a string, the operator shows the string. If a number, the operator adjusts the text position by that amount. The number is expressed in one-one thousandth of a unit in unscaled text space (this is the value T_j above). A positive value has the effect of moving a glyph to the left, and a negative to the right. This is hard to interact with in TeX .

8 Annotations

An annotation is an interactive feature of a PDF, which allows the user to interact with the PDF using their keyboard and/or mouse. In this section we discuss various (but not all) annotation types. A page object has an optional **Annots** field, which is an array of indirect references to all the annotation objects occurring on the page. Annotation objects are dictionaries which represent the annotation on the specified page. An annotation may be referenced from a single page's **Annots** field only; it cannot be referenced by two pages simultaneously.

The fields common to all annotations are

Key	Type	Value
-----	------	-------

Type	name	(Optional) the type of PDF object this dictionary describes; must be Annot .
Subtype	name	(Required) the type of annotation this dictionary describes (see below).
Rect	array	(Required) a rectangle (array of 4 numbers $[x\ y\ w\ h]$) which specifies the location of the annotation on the page.
Contents	string	(Optional) text that shall be displayed for the annotation (if relevant for the type of annotation); or an alternative description of the annotation's contents for accessibility readers.
P	dictionary	(Optional) an indirect reference to the page object associated with this annotation.
AP	dictionary	(Optional) an <i>appearance dictionary</i> which specifies how the annotation should be presented on the page.
AS	name	(Required if the annotation's appearance dictionary contains subdictionaries) the annotation's <i>appearance state</i> , which selects the applicable appearance stream from an appearance subdictionary.
Border	array	(Optional) an array specifying the characteristics of the annotation's border, which is a rounded rectangle. The array may have three or four elements, of the form $[r_h\ r_v\ w\ [o_n\ o_{ff}]]$ (the last element may be omitted). r_h and r_v are the horizontal and vertical radii, respectively. w is the stroke width of the border. $[o_n\ o_{ff}]$ is the dash array of the border, it draws dashes of length o_n with spaces of length o_{ff} (the phase is 0).
C	array	(Optional) an array of numbers in $[0, 1]$ which represent the color to be used for <ul style="list-style-type: none"> • the background of the annotation's icon when closed; • the title bar of the annotation's popup window; • the border of a Link annotation. <p>The number of array elements determines the color space in which the color is to be interpreted.</p> <ul style="list-style-type: none"> 0 no color; transparent; 1 DeviceGray; 3 DeviceRGB; 4 DeviceCMYK;

8.1 Border styles

An annotation may be surrounded by a border, printed within its annotation rectangle, when displayed or printed. Some annotations (see below) may use the **BS** field in place of the **Border** field to specify a *border style dictionary* (instead of an array of numbers), which contains information on how the border should be displayed. The border style dictionary may have the following fields:

Key	Type	Value
Type	name	(Optional) the type of PDF object this dictionary describes; must be Border .
W	number	(Optional) the border width in points. Default value: 1.

S	name	(Optional) the border style: <ul style="list-style-type: none"> • S: (solid) a solid rectangle; • D: (dashed) a dashed rectangle, the dash pattern may be specified by the D entry; • B: (beveled) a simulated embossed rectangle that appears to be raised above the surface of the page. • I: (inset) a simulated engraved rectangle that appears to be recessed below the surface of the page. • U: (underline) a single line along the bottom of the annotation rectangle.
		Default value: S .
D	array	A <i>dash array</i> of the form $[o_n\ o_{ff}]$ which specifies the pattern the dash pattern should take on (on for o_n units, off for o_{ff} units). Default value: $[3]$ (equivalent to $[3\ 3]$).

8.2 Appearance streams

An annotation may also specify *appearance streams*, which form alternatives to the border and color attributes available via the **AP** and **AS** fields. **AP**, the appearance dictionary, may specify the following fields:

Key	Type	Value
N	stream or dictionary	(Required) the annotation’s normal appearance.
R	stream or dictionary	(Optional) the annotation’s rollover appearance. Default: the value of the N entry.
D	stream or dictionary	(Optional) the annotation’s down appearance. Default: the value of the N entry.

The annotation has the appearance determined by **N**; until the user’s cursor hovers over the annotation’s active area, and the appearance changes to **R**; when the user presses down the mouse button within the annotation’s active area, the appearance changes to **D**. The values of each entry may be a stream, or an *appearance subdictionary*. In the latter case, the appearance subdictionary defines multiple appearance streams according to different *appearance states* of the annotation. In the latter case, the **AS** must specify which appearance state is to be displayed by default.

For example, if we have a checkbox, it may have two states: **On** and **Off**. We have appearances for normal and down, but we don’t need one for rollover. Then the appearance dictionary and appearance state may be:

```

1  /AP <<
2    /N <<
3      /On 1 0 R
4      /Off 2 0 R
5    >>
6    /D <<
7      /On 3 0 R
8      /Off 4 0 R
9    >>
10 >>
11 /AS /Off

```

8.3 Text annotations

A text annotation appears as some object on the page (according to its appearance dictionary), and when the user hovers over it, a “sticky note” with specified text on it appears. A text annotation’s dictionary may have the following additional fields:

Key	Type	Value
Subtype	name	(Required) the type of annotation this dictionary describes; must be Text .
Open	boolean	(Optional) a flag specifying whether the annotation (sticky note) should be displayed open initially. Default: false .

Name	name	(Optional) the name of an icon that should be used to display the annotation. This field is only used if the appearance dictionary AP is not specified. Valid names are: Comment , Key , Note , Help , NewParagraph , Paragraph , Insert . Default: Note .
-------------	------	---

For example, take the following annotation:



It can be created using the following code:

```

1 <<
2   /Type /Annot
3   /Subtype /Text
4   /Contents (This text should display when you hover over the icon)
5   /Name /Help
6   /Rect [72 769.89 82 779.89]
7 >>

```

We can also create a text annotation which alters its state depending on the whether the user's cursor is hovering over it, or clicking, or not. First we need to create its three states:

```

1 1 0 obj
2 <<
3   /Type /XObject
4   /Subtype /Form
5   /BBox [0 0 50 50]
6   /FormType 1
7   /Matrix [1 0 0 1 0 0]
8   /Resources 2 0 R
9   /Length 645
10 >>
11 stream
12 q 50 0 0 50 0 0 cm
13 BI
14 /BPC 8
15 /CS /DeviceRGB
16 /F [ /AHx ]
17 /W 9
18 /H 9
19 ID
20 ... % code for smiley
21 EI
22 Q
23 endstream
24 endobj
25 4 0 obj
26 <<
27   /Type /XObject
28   /Subtype /Form
29   /BBox [0 0 50 50]
30   /FormType 1
31   /Matrix [1 0 0 1 0 0]
32   /Resources 5 0 R
33   /Length 645
34 >>
35 stream
36 q 50 0 0 50 0 0 cm
37 BI
38 /BPC 8
39 /CS /DeviceRGB
40 /F [ /AHx ]
41 /W 9
42 /H 9
43 ID
44 ... % code for straight face
45 EI
46 Q
47 endstream
48 endobj
49 6 0 obj
50 <<
51   /Type /XObject
52   /Subtype /Form
53   /BBox [0 0 50 50]

```

```

54 /FormType 1
55 /Matrix [1 0 0 1 0 0]
56 /Resources 7 0 R
57 /Length 645
58 >>
59 stream
60 q 50 0 0 50 0 0 cm
61 BI
62 /BPC 8
63 /CS /DeviceRGB
64 /F [ /AHx ]
65 /W 9
66 /H 9
67 ID
68 ... % code for frowny
69 EI
70 Q
71 endstream
72 endobj

```

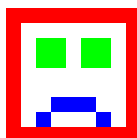
Then our annotation is simply

```

1 % 8 0 obj
2 <<
3   /Type /Annot
4   /Subtype /Text
5   /Contents (This is a face!)
6   /AP <<
7     /N 6 0 R
8     /R 4 0 R
9     /D 1 0 R
10  >>
11   /Rect [72 769.89 122 819.89]
12 >>

```

You can try it out here:



Some annotations will only work on certain PDF viewers. For example, the above annotation will not change states on Sumatra PDF. It will on Adobe Acrobat (except it doesn't seem to support the down state).

There exist other annotation types, like caret annotations and stamp annotations, that do function well with \TeX . But since I don't see a use in them, and similar results can be obtained by text annotations, I will not discuss them here.

8.4 Link annotations

A link annotation represents either a hypertext link to a destination elsewhere in the document, or an action to be performed (explained below). A link annotation may have the following additional entries in its dictionary:

Key	Type	Value
Subtype	name	(Required) the type of annotation this dictionary describes; must be Link .
A	dictionary	(Optional) an action dictionary (see below) describing the action to be performed when the link annotation is clicked.
Dest	array or name or string	(Optional; not permitted if A is present) a destination (see below) that will be jumped to when the link annotation is clicked.

H	name	(Optional) the annotation's <i>highlighting mode</i> , describing the visual effect to be displayed upon clicking the annotation. H may be one of the following: <ul style="list-style-type: none"> • N (none): no highlighting; • I (invert): invert the colors of the contents in the annotation rectangle (default value); • O (outline): invert the color of the annotation's border; • P (push): display the annotation as if it were being pushed down below the surface of the page.
BS	dictionary	(Optional) a border style dictionary (see above) which describes the border (alternative to Border).

8.4.1 Destinations

A destination defines a particular view of a document, and consists of the following:

- (1) the page of the document to be displayed;
- (2) the location on the page to display;
- (3) the magnification (zoom) factor.

When a destination is specified, this can be either given as an *explicit destination* (which is an array), a name (which is defined in the **Dest**s entry of the document catalog), or a string (whose value is defined in the **Dest**s entry of the name dictionary of the document (**Names** entry of the catalog)). In the end these must all resolve to explicit destinations, whose syntax is described below:

Syntax	Meaning
[<i>page</i> / XYZ <i>left top zoom</i>]	Display the page designated by <i>page</i> (a page object), with the coordinates (<i>left</i> , <i>top</i>) positioned in the upper-left corner of the window. The page's contents shall be magnified by a factor of <i>zoom</i> . A null value for any of the parameters (other than <i>page</i>) means that its value will remain unchanged. A <i>zoom</i> of 0 is equivalent to a null value.
[<i>page</i> / Fit]	Display the page designated by <i>page</i> , zoomed so that the page fits within the window in its smaller dimension and centered in the other.
[<i>page</i> / FitH <i>top</i>]	Display the page <i>page</i> zoomed so that the page just fits into the window horizontally, and positioned so that <i>top</i> is at the top of the page.
[<i>page</i> / FitV <i>left</i>]	Like FitH but vertically.
[<i>page</i> / FitR <i>left bottom right top</i>]	Display the page <i>page</i> zoomed so that the rectangle defined by <i>left bottom right top</i> fills the window. If the magnification required for horizontal and vertical dimensions are different, use the smaller zoom factor and center with the other dimension.

8.4.2 Actions

An action is something that the PDF consumer can perform when something is triggered (the mouse clicking on a region, moving over a region, etc.). While actions are quite broad, we will discuss only a few of them and only a few trigger events.

An action is specified by an action dictionary, which may have the following fields (and more, according to the type of action):

Key	Type	Value
Type	name	(Optional) the type of PDF object this dictionary describes; must be Action .

S	name	(Required) the type of action this dictionary describes (see below).
Next	dictionary or array	(Optional) the next action, or actions, to execute after finishing this one.

Essentially each action dictionary forms a tree, whose value is the action to execute and children are the next actions to execute (in order). We discuss the following action types:

Go-To actions

A *go-to action* changes the view to a specified destination. The additional entries to a go-to action dictionary are as follows:

Key	Type	Value
S	name	(Required) the type of action this dictionary describes; must be GoTo .
D	name, string, or array	(Required) the destination to jump to.

Launch actions

A *launch action* opens an application or a document. The additional entries to a launch action dictionary are as follows:

Key	Type	Value
S	name	(Required) the type of action this dictionary describes; must be Launch .
F	string	(Required) the application to launch or document to open.
NewWindow	boolean	(Optional) if the file specified by F is a PDF document, true if it should be opened in a new window, and false if it should open it in the same window.

URI actions

A *uniform resource identifier* (URI) is a string that identifies (resolves to) a resource (either locally or on the Internet). A *URI action* causes a URI to be resolved. The additional entries to a URI action dictionary are as follows:

Key	Type	Value
S	name	(Required) the type of action this dictionary describes; must be URI .
URI	string	(Required) the URI to resolve (encoded in UTF-8).

II. PDFTEX

This article assumes some preliminary knowledge of (plain-) \TeX , so I assume I don't need to introduce \TeX to you. \pdfTeX is a \TeX engine, which converts \TeX code to a PDF (historically, \TeX engines compiled to DVI files). In this part, we will discuss how to use \pdfTeX primitives to manipulate the underlying PDF with more control than standard (Knuth-) \TeX primitives.

The main reference for this section is the \pdfTeX manual; consult it for information regarding primitives not discussed here.

1 \pdfliterals

The most important primitive for our uses that \pdfTeX provides is the \pdfliteral macro. The \pdfliteral primitive inserts a whatsit (explained in the \TeX book, but not really important to our discussion) into the output stream containing raw PDF code. During the shipout of the current box, the PDF code is injected into the current content stream preceeding the whatsit. The exact usage of \pdfliteral is as follows:

\pdfliteral [*shipout*] [*direct* | *page*] *general text*

Where *general text* means material of the form $\{ \langle \textit{balanced text} \rangle \}$ where *balanced text* is text with balanced curly braces. A vanilla \pdfliteral (meaning one without *shipout* or *direct* or *page*) will insert *general text* (which is PDF code) into the document at the current position. It does the following:

- (1) if in a text object, end the text object;
- (2) set the CTM to the current position in the document;
- (3) insert the PDF code.

So for example, the result of

```
1 Some text \pdfliteral{0 0 m 30 0 1 S}\hbox to30bp{\hfil hi!\hfil}
```

Will be the PDF code

```
1 BT
2 /F1 9.9626 Tf 91.925 759.927 Td [(Some)-333(text)]TJ
3 ET
4 1 0 0 1 139.248 759.927 cm
5 0 0 m 30 0 1 S
6 1 0 0 1 -139.248 -759.927 cm
7 BT
8 /F1 9.9626 Tf 148.713 759.927 Td [(hi!)]TJ
9 ET
```

We have here the following:

- (1) a text object, with the text “Some text”;
- (2) a transformation operation (**cm**), which transform CTM to the current position in the document;
- (3) the inserted PDF code;
- (4) a transformation operation, which moves the CTM back to where it was previously.

Normally, *general text* is fully expanded when the whatsit is created. This can be averted by using the *shipout* keyword, which delays expansion to when the box is shipped out.

Furthermore, instead of ending a text object (if in one), if the *direct* keyword is given, \pdfTeX will add the PDF code to the text object. It will also not transform user space (i.e. the code is placed verbatim into the PDF, no matter where). Alternatively, *page* will not transform user space, but it will exit a text object.

By default, units in a PDF are 1/72 of an inch. These are called *big points* in \TeX , abbreviated to **bp**. Normal \TeX pts are 1/72.27 of an inch. So in a \pdfliteral it is often useful to transform user space so that 1 unit = 1pt. This can be done by scaling space by $72/72.27 \approx .996264$.

We define the macro

```
1 \def\pttrans{.996264 0 0 .996264 0 0 cm }
```

and we will use it throughout this part. Note the space after the **cm**, this is because operators must be followed with a space in a PDF. (E.g. you can't do `10 0 0 10 0 0 cm3 w.`)

We also define the following macro which accepts a dimension expression, and expands to its result

without the trailing pt:

```

1 \bgroup\lccode'?'='p\lccode'!='t%
2 \lowercase{\egroup%
3   \def\rmpt#1?!{#1}%
4 }%
5 \def\nopt#1{\expandafter\rmpt\the\dimexpr#1\relax}%

```

So, for example `\nopt{10pt+20pt}` will expand to 30. This is a useful macro when dealing with `\pdf literals` (since their operands don't accept units). This macro uses the trick that `\lowercase` doesn't expand its input, and converts character codes without altering category codes (and ? and ! have category code 12).

So suppose we'd like a macro which draws a dashed box around input text. We could do:

```

1 \def\dashbox#1{\hbox{%
2   \setbox0=\hbox{#1}%
3   \pdfliteral{
4     q
5     \pttrans
6     .3 w
7     [3] 0 d
8     0 -\nopt{\dp0} \nopt{\wd0} \nopt{\dp0+\ht0} re
9     S
10    Q
11  }%
12  \box0%
13 }}

```

This will give, for example:

Heȳā there!

This could of course be done in Knuth-TeX, but with significantly more difficulty

We can also create a macro which draws a circle around text. We will use create an approximate circle with 4 cubic Bézier curves. The optimal control points are each about 0.5523 units away from each endpoint (on a unit circle).

```

1 \def\maxdim#1#2{%
2   \ifdim\the\dimexpr#1\relax>\the\dimexpr#2\relax%
3     \dimexpr#1\relax%
4   \else%
5     \dimexpr#2\relax%
6   \fi%
7 }
8 \def\mult#1#2{\nopt{#1\dimexpr#2pt\relax}}
9 \def\circle#1{
10  #1 0 m
11  #1 \mult{.5523}{#1} \mult{.5523}{#1} #1 0 #1 c
12  \mult{-.5523}{#1} #1 -#1 \mult{.5523}{#1} -#1 0 c
13  -#1 \mult{-.5523}{#1} \mult{-.5523}{#1} -#1 0 -#1 c
14  \mult{.5523}{#1} -#1 #1 \mult{-.5523}{#1} #1 0 c
15  h
16 }
17 \def\circbox#1{\hbox{%
18   \setbox0=\hbox{#1}%
19   \edef\radius{\nopt{.5\maxdim{\wd0}{\ht0+\dp0}}}%
20   \vrule width0pt depth\dimexpr\radius pt +.5\dp0-.5\ht0\relax height\dimexpr\radius pt+.5\ht0-.5\dp0\relax
21   \pdfliteral{
22     q
23     .3 w
24     [3] 0 d
25     1 0 0 1 \nopt{.5\wd0} \nopt{.5\ht0-.5\dp0} cm
26     \circle{\radius} S
27     Q
28   }\box0%
29 }}

```



Now, we can ask ourselves, why is our definition of `\circle` so complicated? Why not just define a parameterless `\unitcircle` which draws a unit circle and then scale it? Because then this will also scale the

width of the line drawn, which will lead to undesirable results. Also of course, this macro isn't perfect: the radius should really be $\sqrt{w^2 + (h + d)^2}$ where w, h, d are the width, height, and depth of the box respectively. But this requires being able to compute the square root, which is beyond the scope of this article.

Finally, the reason for the `\vrule` is to ensure that the box is given the correct height. \TeX is oblivious to the inserted PDF code, it doesn't know its dimensions, so it can't update the box accordingly.

2 XObjects

We now discuss how to utilize XObjects in $\text{pdf}\text{\TeX}$. One may wonder why we don't first discuss PDF objects in general, and the answer is simple: they're less useful than form XObjects in $\text{pdf}\text{\TeX}$. This is because objects are in general much more internal to the PDF, and generally don't display content (unless they're XObjects or the content stream of a page object).

2.1 Form XObjects

$\text{pdf}\text{\TeX}$ allows the creation of a form XObject in two steps:

- (1) create the content of the XObject in a box;
- (2) save the content to a form XObject, while also potentially altering its stream attributes and resources.

The first step is done in the obvious way: by using the \TeX primitive `\setbox`. The second step is done using the $\text{pdf}\text{\TeX}$ primitive `\pdfxform`. Its usage is

`\pdfxform` [*attr spec*] [*resources spec*] *box number*

Where *attr spec* is `attr` *general text*, and *resources spec* is `resources` *general text*.

`\pdfxform` creates a form XObject whose attributes dictionary contain *attr spec*, and resources dictionary contains *resources spec*, and whose stream is determined by the box numbered by *box number*. The PDF object number of the last form XObject created is placed in `\pdflastxform`. You can then paint a form XObject using the `\pdfrefxform` primitive, whose usage is

`\pdfrefxform` *object number*

This essentially does Do to the form XObject specified by *object number*, but it also updates the document's position as well.

\TeX does not deal in generation numbers. That is, all of the generation numbers generated by \TeX are 0. So if an object's object number is N , its object identifier will be N 0.

Recall that form XObjects have a required **BBox** field in their stream dictionary. \TeX calculates this from the dimensions of the box supplied to `\pdfxform`. So if anything protrudes from the box, it will be clipped from the form XObject when painted. This can be annoying, or useful.

Form XObjects are useful for creating recreatable PDF objects, similar to boxes. Unlike boxes, the number of form XObjects is not limited by \TeX . For example, we can create a special symbol point by doing:

```

1 \bgroup
2 \setbox0=\hbox to10.3bp{\vrule width0bp height10.15bp depth.15bp%
3 \kern.15bp%
4 \pdfliteral{
5   1 0 0 rg
6   0 0 0 RG
7   1 j 1 J
8   .3 w
9   0 0 m
10  10 0 1 10 10 1 0 10 1 3 5 1 h B
11 }\hfil}
12 \pdfxform0
13 \xdef\ribbon{\pdfrefxform\the\pdflastxform\relax}
14 \egroup

```

This will create

A ribbon: 

The kerning and the extra dimensions are to take into account the extra space required by rounded joins.

We can use the fact that form XObjects clip their contents to our advantage: we can create infinitely extensible operators using this (like those in my `pdfMsym` package).

```

1 \bgroup

```

```

2 \setbox0=\hbox{$\displaystyle\sum$}
3 \setbox1=\hbox to.52\wd0{\copy0\hss}
4 \setbox2=\hbox to.08\wd0{\kern-.52\wd0\copy0\hss}
5 \setbox3=\hbox to.4\wd0{\kern-.6\wd0\copy0\hss}
6 \pdfxform1 \xdef\suumL{\the\pdflastxform}
7 \pdfxform2 \xdef\suumC{\the\pdflastxform}
8 \pdfxform3 \xdef\suumR{\the\pdflastxform}
9 \egroup
10
11 \def\suum{%
12   \mathop{%
13     \pdfrefxform\suumL%
14     \xleaders\hbox{\pdfrefxform\suumC}\hfill%
15     \pdfrefxform\suumR%
16   }\limits%
17 }

```

What we've done here is create three form XObjects, the first has the left slice of the sum character, which does not extend; the second has the middle slice, which is what we use to extend the operator; and the third has the right slice. These are created by eyeballing the lengths of each slice, and clipping them using form XObjects. We use the fact that the contents of `\mathop` take the width of its limits, and we use leaders to extend the middle slice. This gives, for example:

$$\sum_{nopqrstuvwxyz}^{abcdefghijklmnop}$$

\TeX does not write the XObject into the PDF until it is referred to by `\pdfrefxform`. Thus if you want to paint it yourself using a Do operation, you need to somehow first write it into the PDF. This can be done by prepending `\pdfxform` with `\immediate`. This will write the XObject into memory immediately, instead of waiting for a reference.

2.1.1 Shading

We can also use form XObjects to paint shadings (tiling patterns must be **Pattern** objects, which requires us to use general PDF objects, not XObjects; so they will be discussed in the next section). To do this, we must access the resources of the XObject. So first we create a box with the PDF code we'd like:

```

1 \bgroup
2 \setbox0=\hbox to100bp{\vrule width0bp height50bp depth50bp%
3 \pdfliteral{
4   1 0 0 1 50 0 cm
5   /Pattern cs
6   /Sh scn
7   \circle{50} f
8 }\hfil}

```

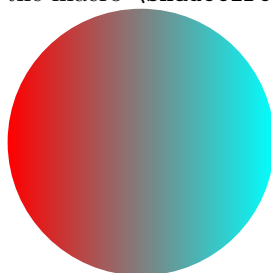
Now we need to define the pattern **Sh** in the **Pattern** subdictionary of the form XObject's resources dictionary:

```

1 \pdfxform resources {
2   /Pattern << /Sh <<
3     /Type /Pattern
4     /PatternType 2 % Shading pattern
5     /Shading <<
6       /ShadingType 2 % axial shading
7       /ColorSpace /DeviceRGB
8       /Domain [0 1]
9       /Coords [0 0 100 0] % the target coordinate space is not affected by changes to the CTM
10      /Function <<
11        /FunctionType 2 % exponential interpolation
12        /Domain [0 1]
13        /C0 [1 0 0]
14        /C1 [0 1 1]
15        /N 1 % linear
16      >>
17    >>
18  >> >>
19 }0
20 \xdef\shadecircle{\pdfrefxform\the\pdflastxform\relax}

```

Now we can draw a shaded circle with the macro `\shadecircle`, which results in:



2.1.2 Transparency

Transparency is also governed by XObjects (transparency group objects), so we will discuss that here. Suppose we want to draw four overlapping circles, each with an opacity of .5. First we draw our circles:

```

1 \bgroup
2 \setbox0=\hbox to100bp{\vrule width0bp height100bp depth0bp%
3 \pdfliteral{
4   /Gs gs
5   .75 g
6   q
7   1 0 0 1 40 60 cm
8   \circle{30} f
9   Q
10  q
11  1 0 0 1 60 60 cm
12  \circle{30} f
13  Q
14  q
15  1 0 0 1 40 40 cm
16  \circle{30} f
17  Q
18  q
19  1 0 0 1 60 40 cm
20  \circle{30} f
21  Q
22 }\hfil}

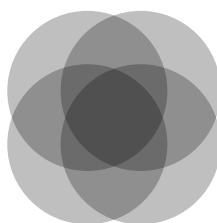
```

Now we add **Gs** to the resources. We will make the fill opacity .5 and the blend mode **Multiply**.

```

1 \pdfxform resources{
2   /ExtGState << /Gs <<
3     /ca 1
4     /BM /Multiply
5   >> >>
6 }0
7 \xdef\fourcircs{\pdfrefxform\the\pdflastxform\relax}
8 \egroup

```



We can also make a more complicated drawing. Say we start with a gradient background, and draw four circles on it with an opacity of .5. But the four circles will form an isolated transparency group, and within the transparency group they are drawn with full opacity.

```

1 \bgroup
2 \setbox0=\hbox to100bp{\vrule width0bp height100bp depth0bp%
3 \pdfliteral{
4   .75 g
5   q

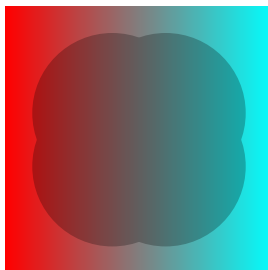
```

```

6      1 0 0 1 40 60 cm
7      \circle{30} f
8      Q
9      q
10     1 0 0 1 60 60 cm
11     \circle{30} f
12     Q
13     q
14     1 0 0 1 40 40 cm
15     \circle{30} f
16     Q
17     q
18     1 0 0 1 60 40 cm
19     \circle{30} f
20     Q
21 } \hfil}
22 \immediate\pdfxform attr{
23   /Group <<
24     /S /Transparency
25     /CS /DeviceGray
26     /I true
27   >>
28 } 0
29 \edef\fourcircsform{\the\pdflastxform}
30
31 \setbox0=\hbox to100bp{\vrule width0bp height100bp depth0bp}%
32 \pdfliteral{
33   /Sh sh
34   /Gs gs
35   /4circs Do
36 } \hfil}
37 \pdfxform resources {
38   /Shading << /Sh <<
39     /ShadingType 2
40     /ColorSpace /DeviceRGB
41     /Coords [0 0 100 0]
42     /Domain [0 1]
43     /Function <<
44       /FunctionType 2
45       /Domain [0 1]
46       /C0 [1 0 0]
47       /C1 [0 1 1]
48       /N 1
49     >>
50   >> >>
51   /ExtGState << /Gs <<
52     /ca 1
53     /BM /Multiply
54   >> >>
55   /XObject <<
56     /4circs \fourcircsform\space0 R
57   >>
58 } 0
59 \xdef\fourcircs{\pdfrefxform\the\pdflastxform\relax}
60 \egroup

```

This creates the following:



2.2 Image XObjects

Say we want to include an image in a T_EX file. We could use an inline image for images that we know the samples of, or an image XObject by defining it explicitly (using `\pdfobj`). But say we have an image file we'd like to include. We can use pdfT_EX primitives for this.

The main character of the show is `\pdfximage`, which reads an image file into memory. It supports the following formats: JPEG (`.jpg` or `.jpeg`), JBIG2 (`.jbig2` or `.jb2`), PNG (`.png`), or PDF (`.pdf`). Its usage is

`\pdfximage` [*rule spec*] [*attr spec*] [*page* *number*] *general text*

general text is the name of the file to import.

The dimensions of the image can be controlled via *rule spec*. If some dimensions (but not all) are given, the others will be scaled to yield the same ratio between width and height + depth as the image's ratio between height and depth. If no *rule spec* is given, the image will have its natural dimensions.

If *general text* refers to a PDF file, *page* *number* may be specified, which imports that specific page from the PDF file.

attr spec defines additional attributes which will be added to the image XObject.

An image imported by `\pdfximage` is not written to the PDF unless `\pdfximage` is preceded by `\immediate`, or `\pdfrefximage` is used on the object number of the image XObject (which is returned by `\pdflastximage`).

3 PDF Objects

To create a PDF object, we use the `\pdfobj` primitive. Its usage is

`\pdfobj` *reserveobjnum* | [*useobjnum* *number*] [*stream* [*attr spec*]] *general text*

If *reserveobjnum* is given, an object number is reserved, which can be accessed by `\pdflastobj`. After doing *reserveobjnum*, you can do *useobjnum* to create a PDF object with that object number.

If *stream* is specified, then the object created is a stream object, whose stream dictionary is specified by *attr spec*.

The object is kept in memory, and not written to the PDF file immediately, until its object number is passed to `\pdfrefobj` or if `\pdfobj` is preceded by `\immediate`. Unlike form XObjects, `\pdfrefobj` does not paint anything, it simply writes the PDF object to the file at shipout time.

The object number can be accessed by `\pdflastobj` after `\pdfobj`.

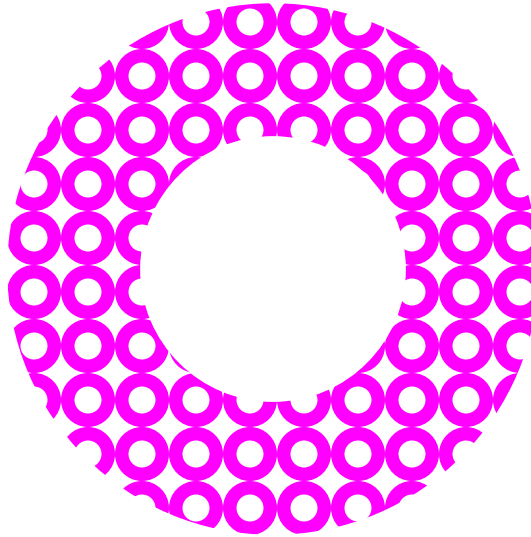
3.1 Tiling patterns

Recall that tiling pattern's pattern cell is defined by a stream object of type **Pattern**. So for example, if we want to create an uncolored tiling pattern whose pattern cell is a circle. We first create the pattern cell, which is just a hollowed out circle:

```
1 \immediate\pdfobj stream attr{
2   /Type /Pattern
3   /PatternType 1
4   /PaintType 2
5   /TilingType 1
6   /BBox [0 0 20 20]
7   /XStep 20
8   /YStep 20
9   /Resources << >>
10 }{
11   q
12   1 0 0 1 10 10 cm
13   \circle{10}
14   \circle{5} f*
15   Q
16 }
```

Now we create the outline of the pattern:

```
1 \bgroup\setbox0=\hbox to200bp{\vrule width0bp height200bp depth0bp%
2 \pdfliteral{
3   q
4   \pttrans
5   1 0 0 1 100 100 cm
6   /CSP cs
7   1 0 1 /P scn
8   \circle{100}
9   \circle{50} f*
10  Q
11 } \hfil}
12 \pdfxform resources{
13   /ColorSpace << /CSP [ /Pattern /DeviceRGB ] >>
14   /Pattern << /P \the\pdflastobj\space 0 R >>
15 } 0
16 \xdef\donutdonut{\pdfrefxform\the\pdflastxform\relax}
17 \egroup
```



3.2 PostScript functions

PostScript functions (type 4 functions) must be written in content streams, and thus require a stream PDF object. For example, suppose we want to create a functional shading which has a weird effect. We define the function $f: [0, 1]^2 \rightarrow [0, 1]$ by

$$f(x, y) = \frac{1 + \sin(1800(x - .5)^2) \sin(1800(y - .5)^2)}{2}$$

(Trigonometric functions are computed in degrees.) Then our shading will be determined by f . To define the function we do:

```

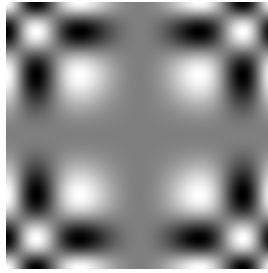
1 \immediate\pdfobj stream attr{
2   /Type /Function
3   /FunctionType 4
4   /Domain [0 1 0 1]
5   /Range [0 1]
6 }{{
7   .5 sub
8   exch
9   .5 sub
10  dup
11  mul
12  1800 mul
13  sin
14  exch
15  dup
16  mul
17  1800 mul
18  sin
19  mul
20  1 add
21  2 div
22 }}
```

Then the actual shading is done by

```

1 \bgroup
2 \setbox0=\hbox to100bp{\vrule width0bp height100bp depth0bp%
3 \pdfliteral{
4   /Sh sh
5 }\hfil}
6 \pdfxform resources {
7   /Shading << /Sh <<
8     /ShadingType 1
9     /ColorSpace /DeviceGray
10    /Domain [0 1 0 1]
11    /Matrix [100 0 0 100 0 0]
12    /Function \the\pdflastobj\space0 R
13  >> >>
14 }0
15 \egroup
```

This creates



4 Annotations

We can add annotation objects using the `\pdfannot` primitive. Its usage is

```
\pdfannot reserveobjnum | [useobjnum <number>] [<rule spec>] <general text>
```

Where *rule spec* is (width | height | depth) <dimen> [<rule spec>]. This should illustrate the reason for using `\pdfannot` over `\pdfobj` and with `/Type /Annot`: you can give the annotation's activation area.

Everything here is pretty self-explanatory. We can create a text annotation with different states like so:

```
1 \bgroup
2 \setbox1=\hbox{Rollover}
3 \setbox0=\hbox to\wd1{\hfil Normal\hfil}
4 \setbox2=\hbox to\wd1{\hfil Down\hfil}
5 \immediate\pdfxform0 \edef\normal{\the\pdflastxform}
6 \immediate\pdfxform1 \edef\rollover{\the\pdflastxform}
7 \immediate\pdfxform2 \edef\down{\the\pdflastxform}
8 \pdfannot width\wd0 height\ht0 depth\dp0 {
9   /Subtype /Text
10  /Contents (This is a text annotation with different states)
11  /AP <<
12    /N \normal\space0 R
13    /R \rollover\space0 R
14    /D \down\space0 R
15  >>
16  /Border [3 3 1]
17  /C [0 1 1]
18 }
19 \egroup
```

This creates:

Normal

Many PDF viewers don't support multiple-state annotations. Try viewing this PDF on Adobe Acrobat.

4.1 Links and destinations

Consider the following issue: suppose we have an explicit destination we want to jump to. But the link itself spans multiple lines. \TeX will break the link over multiple lines, but how should the link border look? Well pdf \TeX actually provides a primitive for creating link annotations so that they can break across lines or even pages. It does this by inserting a new annotation for each line.

This is done via the primitives `\pdfstartlink...``\pdfendlink`, whose usage is

```
\pdfstartlink [<rule spec>] [attr <general text>] <action spec>
```

where *action spec* is

```
user <general text> | goto <goto action spec>
```

where *goto action spec* is

```
name <general text> | page <number> <general text>
```

attr spec defines additional attributes for the annotation (e.g. **Border**, **C**). *action spec* defines the action to be done when the annotation's activation region is clicked. If *action spec* is **user**, *general text* is added directly to the annotation's dictionary. We will discuss **goto** actions later. For example, we may have a URL hyperlink:

```
1 \pdfstartlink
2   attr {
3     /Border [3 3 1]
4     /C [1 0 0]
5   }
```

```

6   user {
7     /Subtype /Link
8     /A <<
9       /S /URI
10      /URI (https://tug.org/)
11    >>
12  }
13  \TeX{} User Group%
14  \pdfendlink

```

This creates

TeX User Group

If the *goto action spec* is **page** $\langle number \rangle$, when clicked, the document goes to the page specified by *number*, and the magnification is given by *general text* (**/Fit**, etc.).

4.1.1 Destinations

You can create an indirect destination using the primitive `\pdfdest`. Its usage is

```
\pdfdest name  $\langle general\ text \rangle$  (xyz [zoom  $\langle number \rangle$ ] | fitr  $\langle rule\ spec \rangle$  | fith | fitv | fit)
```

This creates a destination at the current point whose name is *general text*. The type of destination is specified by the rest of the input (e.g. **xyz** for **/XYZ**). This destination can be referred to by its name as a string; its name is put in the name tree defined by the **Dests** subdictionary of the document catalog's **Name** dictionary.

If **xyz** is specified, an optional **zoom** argument can be given. A zoom of 1000 gives the normal page view.

You can then then create a link to this destination like so:

```

1  \pdfdest name {link name} xyz zoom 2000
2  ...
3  \pdfstartlink
4    attr {
5      /Border [3 3 1]
6      /C [1 0 0]
7    }
8    goto name {link name}%
9  This is a link%
10 \pdfendlink

```

5 Graphic State Stacks

5.1 Color stacks

Consider the following code to color some text:

```
1 \quitvmode\pdfliteral{q 1 0 0 rg 1 0 0 RG}Hello\pdfliteral{Q}Hey!
```

(`\quitvmode` is a pdfTeX primitive which has essentially the same effect as `\leavevmode`.) This will result in the following PDF code:

```

1 1 0 0 1 91.925 759.927 cm
2 q 1 0 0 rg 1 0 0 RG
3 1 0 0 1 -91.925 -759.927 cm
4 BT
5 /F1 9.9626 Tf 91.925 759.927 Td [(Hello)]TJ
6 ET
7 1 0 0 1 114.341 759.927 cm
8 Q
9 1 0 0 1 -114.341 -759.927 cm
10 BT
11 /F1 9.9626 Tf 114.341 759.927 Td [(Hey!)]TJ
12 ET

```

Let's see what happens here:

- (1) the CTM is translated so that it points at the current position on the page;
- (2) the graphics state is pushed onto the stack, and the color is changed to red;
- (3) the CTM is translated back (since after the `\pdfliteral` is executed, the CTM must be repositioned);
- (4) "Hello" is printed;
- (5) the CTM is translated to the current position for the next `\pdfliteral`;

- (6) the `\pdfliteral` injects `Q` to the content stream;
- (7) the CTM is translated back;
- (8) “Hey!” is printed.

Note the issue: `Q` is injected into the content stream, which pops the last graphics state. This graphics state was had the CTM positioned at `[1 0 0 1 91.925 759.927]`, the placement of “Hello”. So the “Hey!” will be printed *over* “Hello”, instead of after it!

So we need some way of letting `TEX` handle the graphics state stack. This can be done via two new pdf`TEX` primitives: `\pdfcolorstackinit` and `\pdfcolorstack`.

`\pdfcolorstackinit` initializes a `TEX` graphics state stack (also called a color stack), which can then be manipulated by `\pdfcolorstack`. The usage of `\pdfcolorstackinit` is as follows:

```
\pdfcolorstackinit [page] [direct] <general text>
```

this initializes a `TEX` graphics state stack, whose initial value is determined by *general text*. A usage of `\pdfcolorstackinit` expands to the number of the `TEX` graphics state stack, which is used by `\pdfcolorstack` to manipulate the stack. If *page* is given, then the `TEX` graphics state stack is restored at the beginning of each page. If *direct* is given, operations by `\pdfcolorstack` will not break the current text object in which they may reside.

The usage of `\pdfcolorstack` is as follows:

```
\pdfcolorstack <stack number> (set | push | pop | current) <general text>
```

This manipulates the stack given by *stack number* according to the operation specified:

- **set**: the current value of the `TEX` graphics stack is replaced with *general text*;
- **push**: *general text* is pushed to the top of the `TEX` graphics stack, and becomes the new current value;
- **pop**: the top value of the `TEX` graphics state is removed and the new top becomes the current graphics state;
- **current**: the PDF graphics state is set to the top value of the `TEX` stack without modifying the `TEX` graphics state stack.

Importantly, `TEX` implements its color stacks without using the `q.Q` operators. When `\pdfcolorstack` is used, it simply puts the top of the stack into the PDF content stream. For example:

```
1 \chardef\tgs=\pdfcolorstackinit direct {0 g 0 G}
2
3 \pdfcolorstack\tgs push{1 0 0 rg 1 0 0 RG}Hi
4 \pdfcolorstack\tgs pop{}there!
```

Will result in the PDF code

```
1 1 0 0 rg 1 0 0 RG
2 BT
3 /F1 9.9626 Tf 91.925 759.927 Td [(Hi)]TJ
4 0 g 0 G
5 [-333(there!)]TJ
6 ET
```

Notice that instead of wrapping the first color change and text object in `q...Q`, it simply places `0 g 0 G` (the head of the stack after the `pop`) after it. A `TEX` graphics state can be used for any graphics state parameter, but it should be used for the same set of parameters for each `push` operation. That is, if you start with the stack initialized to `0 g 1 Tr`, don't all of the sudden push `0 G`. The reason being that when you `pop`, you won't reset the stroking color (because `0 g 1 Tr` doesn't change the stroke color). Similarly, don't *just* push `1 g`, since if you then push something which uses `Tr`, popping won't reset it.

5.2 The issue with transformations

Another issue is that of transformations. Suppose we do

```
1 \chardef\tgs=\pdfcolorstackinit{1 0 0 1 0 0 cm} % cant change CTM in a text object;
2 % cannot be direct
3
4 Hi
5 \pdfcolorstack\tgs push{2 0 0 2 0 0 cm}there,
6 \pdfcolorstack\tgs pop{}pal!
```

The resulting PDF code shouldn't be surprising:

```

1 BT
2 /F1 9.9626 Tf 91.925 759.927 Td [(Hi)]TJ
3 ET
4 1 0 0 1 105.486 759.927 cm
5 2 0 0 2 0 0 cm % here
6 1 0 0 1 -105.486 -759.927 cm
7 BT
8 /F1 9.9626 Tf 105.486 759.927 Td [(there,)]TJ
9 ET
10 1 0 0 1 133.741 759.927 cm
11 1 0 0 1 0 0 cm % here
12 1 0 0 1 -133.741 -759.927 cm
13 BT
14 /F1 9.9626 Tf 133.741 759.927 Td [(pal!)]
15 ET

```

Notice that when we pop from the \TeX graphics state stack, it just writes `1 0 0 1 0 0 cm` to the PDF file. Which effectively does nothing. So we're left with:

Hi there, pal!

Now suppose we use `\pdfliterals` instead:

```

1 Hi
2 \pdfliteral{q 2 0 0 2 0 0 cm}there,
3 \pdfliteral{Q}pal!

```

Then we get the following result:

Hi there
pal

Which is produced by the following PDF code:

```

1 BT
2 /F1 9.9626 Tf 91.925 759.927 Td [(Hi)]TJ
3 ET
4 1 0 0 1 105.486 759.927 cm % position for "there,"
5 q 2 0 0 2 0 0 cm % insertion here
6 1 0 0 1 -105.486 -759.927 cm % move back
7 BT
8 /F1 9.9626 Tf 105.486 759.927 Td [(there,)]TJ
9 ET
10 1 0 0 1 133.741 759.927 cm % position for "pal!"
11 Q % insertion here
12 1 0 0 1 -133.741 -759.927 cm % move back
13 BT
14 /F1 9.9626 Tf 133.741 759.927 Td [(pal!)]TJ
15 ET

```

Once again we can see that the issue is we reset the state to the position where “there,” is to be printed.

While pdf \TeX does provide the primitives `\pdfsetmatrix`, `\pdfsave`, and `\pdfrestore`, these do little but make \TeX aware of the changes to the CTM so it can properly place link and anchor positions. They don't do much else (`\pdfsave` injects `q` into the content stream, `\pdfrestore` injects `Q`).

An important note regarding transformations

When changing the CTM, make sure that the `q` and `Q` are injected into the PDF at the same *place* in the document spatially. Otherwise, as you can see above this will mess up the CTM since the translation back is after the `Q`. So the result is that the CTM is translated back (in the above example) -133.741 bps. Thus when you wrap a CTM change in `q...Q`, also wrap it in `\rlap`, like so:

```

1 Hi,
2 \pdfliteral{q 2 0 0 2 0 0 cm}\rlap{there,}%
3 \pdfliteral{Q}pal!

```

So while we still overlap with “there”, we don't mess subsequent text (which will happen if the CTM is messed up). The resulting PDF code is:

```

1 BT
2 /F1 9.9626 Tf 91.925 759.927 Td [(Hi)]TJ
3 ET
4 1 0 0 1 105.486 759.927 cm % this matches with
5 q 2 0 0 2 0 0 cm
6 1 0 0 1 -105.486 -759.927 cm

```

```
7 BT
8 /F1 9.9626 Tf 105.486 759.927 Td [(there,)]TJ
9 ET
10 1 0 0 1 105.486 759.927 cm
11 Q
12 1 0 0 1 -105.486 -759.927 cm % this
13 BT
14 /F1 9.9626 Tf 105.486 759.927 Td [(pal!)]TJ
15 ET
```