CityScape Data Structure

Version 12.10
February 3, 1998

Table of Contents

1. Introduction

This document defines the data structures and file formats used by Grand Theft Auto.

1.1 Scope

This document is intended to be read by programmers involved in the development of game code and related tools for Grand Theft Auto

1.2 References

[1] **Grand Theft Auto Mission Specification**
- DMA Design Ltd

2. Map Data

2.1 Array

A data structure will be required to store the block information.
This will be in the form of a 3D array, with each element containing face graphic & block type information. The size of the array will be fixed. It will be 256x256x6.

```
typedef struct {
        UInt16 type_map;
        UInt8 type_map_ext;
        Char left, right, top, bottom, lid;
} block_info;

block_info city_scape[6][256][256];
```

type_map :
this is a bitmap with the following structure:

bits	content
bits 0-3	direction bits 0 - up OK (1 = yes, 0 = no) 1 - down OK (1 = yes, 0 = no) 2 - left OK (1 = yes, 0 = no) 3 - right OK (1 = yes, 0 = no)
bits 4-6	block type (0 = air, 1 = water, 2 = road, 3 = pavement, 4 = field, 5 = building, 6 = unused, 7 = unused)
bit 7	flat (1 = yes, 0 = no)
bits 8-13	slope type (0 = none 1- 2 = up 26 low, high 3 - 4 = down 26 low, high 5 - 6 = left 26 low, high 7 - 8 = right 26 low, high 9 - 16 = up 7 low - high 17 - 24 = down 7 low - high 25 - 32 = left 7 low - high 33 - 40 = right 7 low - high 41 - 44 = 45 up,down,left,right)
bit 14-15	rotation of lid (0 = normal, 1 = 90□, 2=180□, 3=270□)

type_map_ext:
this is a bitmap with the following structure:

bits	content
bits 0-2	traffic light bits 1,2,3 (3 bit code for traffic light info)
bits 3-4	remap (0, 1, 2, 3)
bit 5	flip top & bottom faces (1 = yes, 0=no)
bit 6	flip left & right faces (1 = yes, 0=no)
bit 7	railway (1 = yes, 0 = no)

Notes :
(4-6) Roads are where road vehicles tend to go

(4-6) Pavements are where pedestrians tend to go
(4-6) Fields are where neither road vehicles nor pedestrians tend to go
(4-6) Water is where water vehicles must go
(0-6,8,15,x0-2, x3) The road, water, field, pavement, direction, railway & traffic light bits are set in the block above the one which actually stores the graphic for the feature. This means that the very top layer cannot be used for road, water, pavement, etc. - see 2.2
(8) - Railway can go over road, pavement or field.
(0-3) Direction bits are normally used on road. They define the direction which computer-controlled cars are allowed to go in on road squares. Direction bits in air are used to represent the starting position for computer-controlled vehicles.
(7) Flats are straight up squares which are drawn only at the left and top of block spaces. The opposite sides are drawn at the same position. The blocks are drawn transparently. This is intended to be used for roadsigns, fences, etc. The block can be walked/driven through. If a lid is present then it is drawn transparently also.
(x3-4) remapping applies to lids only. A value of zero means no remap. 1-3 means look up tile remap index (in style, see 3.4 remap_index) to get remap table number. Lid is then drawn using that remap.
(x5-6) flip means that the indicated faces are drawn with a left-right flip

left, right, top, bottom, lid:
these bytes store the face information - each byte stores a value which indicates the correct graphic square to use for that face. Multiply by 64x64 to get the offset into the face graphics file. For **left**, **right**, **top** and **bottom**, this is the block side graphics. For **lid**, it is the block lid graphics (see 3.1 Block Data). A value of zero indicates no face - hidden faces must be set to zero.

2.2 Above Types

Some type information is required to be in the block above the graphic which actually represents it. This is necessary to limit the number of blocks which have to be checked by the game engine during play. The editor, however, will not actually place the information in the above blocks - instead, the conversion program, **MapCom**, will move the relevant bits to the correct places before doing any compression. This has the effect that no information can be stored in the editor for blocks which have road, etc as the lid. Instead, the conversion program will decide whether they are air or building based on whether or not they have any sides.

2.3 Array Order

Using [z][y][x] order, position [0][0][0] is the back left top corner of the world.
[5][n][n] is the lowest level. Note that at this level only lids are visible.

2.4 Map Compression

The map data is converted to a compressed format before being loaded by the game, to reduce memory usage.

After compression, the map is stored in 3 data areas :

UInt32 **base** [256][256];
UInt16 **column**[];
block_info **block**[];

base
- a 2D array of 32-bit unsigned ints, where each int stores a byte offset into **column**. - i.e. it stores a pointer to the column which sits at that square on the ground.

column
- a variable length array of shorts (16-bit unsigned integers). For each column, the format is:

```
typedef struct {
        Uint16 height;
        UInt16 blockd[];
} col_info;
```

Here, **height** is the minimum array index for the column (i.e. 6-N where N is the number of blocks in the column). Note that this height may include a top block which has a type but no graphic.
blockd is a variable length array of block numbers, with **blockd**[0] being the ground-level one. Each block number is a reference to the block data stored in **block**.

block
- a variable length array of **block_info** structures, containing every distinct combination of faces & types required for the level.

2.5 object_pos

Object position data is stored in a list which is appended to the compressed map data. The data stored for each object position is :

```
typedef struct {
        UInt16 x,y,z;
        Uint8 type;
        UInt8 remap;
        Ang16 rotation;
        Ang16 pitch;
        Ang16 roll;
} object_pos_struct;
```

Here, (**x**,**y**,**z**) is the position of the object in the world, stated in world co-ordinates (but not fixed point). **type** is the object type - a code between zero and the maximum number of object types which gives an index into the object info (see 3.1.2). **remap** is a remap table number, indicating a remap which is to be applied to this object (0 for none). If **remap** is >=128 then the item is a car, not an object (see 2.5.2 Car Positions).

rotation, **pitch** and **roll** are the initial rotation, pitch and roll angles of the object. They are of type **Ang16**, where this is a two-byte positive integer between zero and **MAX_ANGLE**, where **MAX_ANGLE** is the equivalent of 360□.
There is one entry in this list for each distinct object which is to be present in the world when the game starts.

2.5.1 Traffic Light Boxes

Traffic light boxes are a special type of object. They cannot be rotated. Their rotation value, instead, contains a roadblock route number. This specifies which roadblock route is relevant to the junction at which this traffic light box sits. See 2.6.1 Roadblock Information.

2.5.2 Car Positions

Initial car positions are also stored in **object_pos**. When reading through the **object_pos** data, cars are recognized by the fact that they have a **remap** value of >=128. Car records can be treated as the following structure:

```
typedef struct {
        UInt16 x,y,z;
        UInt8 type;
        UInt8 remap;
        UInt16 rotation;
        UInt16 unused – must be 0;
        UInt16 unused – must be 0;
} car_pos_struct
```

Here, **type** is the car number, an index into **car_info** (see 3.1.3) **x**, **y**, **z** and **rotation** have the same meaning as with objects.
remap is the remap table to be used for this car + 128.

2.5.3 limits

The maximum number of objects (not including cars) which can be loaded by the game is 2500. The

maximum number of cars which can be loaded by the game is 250. The maximum number of data items in an **object_pos** section is 2750.

2.6 route

Routefinder information is also appended to the compressed map data. A doubleword in the header of the compressed map file gives the size in bytes of the complete route data.

An individual route is a list of between 1 and 50 vertices, where each vertex has the following format :

```
typedef struct {
        Uint8 x, y, z;
} file_vertex_struct;
```

A **.rop** file contains the information for one police route (i.e. a variable length array of **file_vertex_struct** structures). The **route** part of the compressed map file can contain any number of police routes. It is of the format :

```
UInt8 num_vertices;                )
UInt8 route_type;          )
file_vertex_struct vertex_list [];     )  repeated for each route
```

where **route_type** is 255 for police.

2.6.1 Roadblock Information

As well as the police routes, the **route** area can also contain roadblock routes. These are variable length lists of co-ordinates of junctions which are relevant to some particular roadblock. Each roadblock route is numbered. There can be up to 254 of them per map (0-253). This number is referenced from within the object position info (see 2.5.1 Traffic Light Boxes).
Roadblock routes are stored in **xxx**.**prb** files, where **xxx** is the roadblock route number. In the **route** area, this number is stored in **route_type**. Apart from that, the data format is the same as for police routes.

2.7 location data

Also in the map file can be found location data. This is in the following format :

```
typedef struct {
        file_vertex_struct police_station [6];
        file_vertex_struct hospital [6];
        file_vertex_struct unused [6];
        file_vertex_struct unused [6];
        file_vertex_struct fire_station [6];
        file_vertex_struct unused [6];
} location_data_struct;
```

This stores (x,y,z) location information for the accident services bases. Unused bases will have a location of (0,0,0). Note that this means that no base can actually be at (0,0,0). There must be at least one and no more than six bases for each service.

2.8 Navigational Data

Navigational data is also stored in the map file. This consists of the name of each area of the map, where it is, and how to say it. Each named area must be a rectangle. Overlapping of areas is allowed - the smaller area always has priority. Every block of the map must be be within at least one named area. A sequence of structures of the following format is stored in the **nav_data** area of the map file :

```
typedef struct {
        UInt8 x,y;                  // top left of the rectangle, array co-ords
```

```
    Uint8 w,h;          // width & height of the rectangle, in blocks
    UInt8 sam;          // sample number to say the name
    Char name[30];      // null-terminated string - text of the name
} nav_data_struct
```

Also relevant to the navigational data is the sample file number, stored in the map file header. This defines which sample file is to be loaded with this map - all sample numbers in the nav_data refer to this sample file.

2.9 File Format

The compressed map data, as described above, is stored in a file **XXX.cmp**. The format of this file is as follows:

name	size	notes
version code	UInt32	currently 331 - note that this number must also be at the start of the uncompressed map file
style number	UInt8	style number xxx means that the style file to be used for this map is stylexxx.grx
sample number	UInt8	sample file number xxx means that the sample file to be used for this map is samxxx.sam
reserved	Uint16	
route_size	UInt32	no. bytes of route data
object_pos size	UInt32	no. bytes of object_pos data
column_size	UInt32	
block_size	UInt32	
nav_data_size	UInt32	no. bytes of nav_data (navigational data)
base	256x256x4	
column	column_size	
block	block_size	
object_pos	object_pos_size	
route	route_size	
location_data	3x6x6	
nav_data	nav_data_size	

3. Style Data

3.1 Block Data

The block face data is stored in 3 areas :

Char ***side_block**;
Char ***lid_block**;
Char ***aux_block**;

Each of these stores between 0 and 256 64x64 block faces (in raw format). The total number of faces among all 3 is limited to 384. (e.g. 224 + 96 + 64)
side_block stores the side block faces.
lid_block stores the lid block faces.
aux_block stores the auxilliary block faces (for animation).

3.1.1 Synchronous Animation

The animation data is stored in 2 areas:
Char ***anim**;
list_anim ***list**;

anim
- a single-byte count, the number of animated blocks, followed by that number of **loaded_anim** structures:

```
typedef struct  {
        UInt8 block;
        UInt8 which;
        UInt8 speed;
        UInt8 frame_count;
        UInt8 frame[];
} loaded_anim;
```

where we have:
block - the block number in question
which – the area type (0 for side, 1 for lid)
speed - the number of game cycles to display each frame
frame_count - the number of auxilliary frames
frame[] - an array of block numbers -> these refer to **aux_block**.

list
- an array **list_anim** structures:

```
typedef struct {
        UInt8 counter;
        UInt8 current_frame;
        loaded_anim *loaded;
} list_anim;
```

where we have:
counter - used for counting game cycles
current frame - used to store the curently visible frame number
loaded - a pointer to the relevant **loaded_anim** entry.

When displaying the animation, the original **side/lid_block** face is shown first, then it is replaced by each of the specified **aux_block** faces in turn, until the animation repeats by going back to the first frame.
Note that **list** is generated internally by the game only.

3.1.2 Object Info

Object type description information is stored in a list which is appended to the block data on disk. For each object type, the following variable-length information record is stored :

```
typedef struct {
        Fix32 width, height,depth;
        UInt16 spr_num;
        UInt16 weight;
        UInt16 aux;
        Int8 status;
        UInt8 num_into;
        UInt16 into[MAX_INTO];
} object_info_struct;
```

Here, **width**, **height** and **depth** store the dimensions of the object with respect to collision checking. Note that **width** is for x, **height** is for y and **depth** is for z. These are Fix32 values but only the high end is stored in the file - this is shifted to make a correct Fix32 when the data is loaded. Hence, the value in the file is simply a pixel count (to the resolution of 64 pixels per block).
spr_num is the first sprite number offset for this object. Note that this number is relative to the first object sprite in the sprites file. However, it is converted to an absolute sprite number on loading. Subsequent sprites for this object will be found at subsequent sprite numbers. There will normally be 15 sprites for each object.
weight is a weight descriptor for the object.
status is a status descriptor for the object, which determines how it behaves. The meanings of the status descriptor are :

status	meaning
0	normal
1	ignorable (can drive over)
2	smashable (breaks on landing)
3	invisible

num_into is how many other objects this object breaks into when damaged.
into is a list of **num_into** object type codes, defining the objects which this object can break into.
aux is an auxilliary word whose meaning depends on the type of object.

3.1.2.1 Animated Objects

Animated objects are a special case. They cannot be involved in collisions and are there for graphical effect only. The same data structure is used, with the following differences :

- **height** stores the number of game cycles per frame
- **width** stores the number of frames
- **depth** stores a life descriptor (0 for infinite, non-zero n for n animation cycles)

The animation works by displaying each of the desired frames in turn for the desired number of game cycles, then returning to the first frame after the last one.
If the life descriptor is set to 0 then the animation repeats indefinitely. If it is set to a non-zero number then the animation is played for that number of times and then the object destroys itself.

3.1.3 Car Info

Car type information is stored in a list which is also stored in the style file. The following type definitions are used :

```
typedef struct {
        Int16 rpx, rpy;
        Int16 object
```

```
        Int16 delta
} door_info_struct;

typedef struct {
        Int16 width, height, depth;
        Int16 spr_num;
        Int16 weight;
        Int16 max_speed, min_speed;
        Int16 acceleration, braking;
        Int16 grip, handling;
        hls_info_struct remap24[12];
        UInt8 remap8[12];
        UInt8 vtype;
        UInt8 model;
        UInt8 turning;
        UInt8 damagable;
        UInt16 value[4];
        Int8 cx,cy;
        Int32 moment;
        float rbp_mass;
        float g1_thrust;
        float tyre_adhesion_x, tyre_adhesion_y;
        float handbrake_friction;
        float footbrake_friction;
        float front_brake_bias;
        Int16 turn_ratio;
        Int16 drive_wheel_offset;
        Int16 steering_wheel_offset;
        float back_end_slide_value;
        float handbrake_slide_value;
        UInt8 convertible;
        UInt8 engine;
        UInt8 radio;
        UInt8 horn;
        Uint8 sound_function;
        Uint8 fast_change_flag;
        Int16 doors;
        door_info_struct door [MAX_DOORS]
} car_info_struct;
```

Here, **width**, **height** and **depth** store the dimensions of the car with respect to collision checking. Note that **width** is for x, **height** is for y and **depth** is for z. The value in the file is simply a pixel count (to the resolution of 64 pixels per block).
spr_num is the first sprite number offset for this car. Note that this number is relative to the first car sprite in the sprites file. However, it is converted to an absolute sprite number on loading.
weight is a weight descriptor for the car.
max_speed and **min_speed** are descriptors of the maximum and minimum possible speeds for the car.
acceleration, **braking**, **grip**, **handling** and **turning** are descriptors of these characteristics of the car.
damagable reflects how easily the car can sustain damage
vtype is a descriptor of the type of car / vehicle. The meanings are :

vtype	meaning
0	bus
1	front of juggernaut
2	back of juggernaut
3	motorcycle
4	standard car

8	train

model is a sub-type within **vtype** for cars which holds an identifier for the model of car.
remap (8 - bit) is an array of 12 remap numbers (for 8 bit) and hls remap infos (for 24 bit).
Remaps 1-6 are used for sprayshops. Remaps 7-12 are used for randomly generated dummy cars of this type (along with the default remap which is stored with the car).
value is the monetary value of the car in the GTA mission, in 1000s of $. There are 4 **value** entries for the 4 cranes.
cx, **cy** is the pixel co-ordinates of the centre of mass of the car, relative to the graphical centre.
moment is the moment of inertia of the car.
rbp_mass is the total mass of the car.
g1_thrust is the ratio for 1^{s} gear (only one gear now).
handbrake_friction is the friction of the handbrake.
footbrake_friction is the friction of the footbrake.
front_brake_bias is the front bias of braking.
turn_ratio is the turn ratio of the car.
drive_wheel_offset, **steering_wheel_offset**, **back_end_slide_value** and **handbrake_slide_value** are more handling controls.
engine is the engine type of the car (for sound effects).
convertible is 1 if the car is a convertible, else 0.
radio is the radio listening type of the car.
horn is the horn type of the car.
sound_function and **fast_change_flag** are for audio information.

num_doors is the number of opening doors which this car has. **door** is then a list of **num_doors** door info structures.
For each door, (**rpx**,**rpy**) is the relative position where a pedestrian must stand to enter / exit the car via that door. **delta** is the delta number of that door. **object** is the object type number of the door - this refers to an object info structure (see 3.1.2).

3.1.4 floats

Note that all of the **float** quantities in the car data are stored in the file as 32-bit fixed point values (with 16 bits after the point). This is to aid Playstation compatibility. They are converted to **float** when they are loaded into the game.

3.2 Sprites

3.2.1 Sprite Info

The sprite info part of the style file contains num_sprites variable-sized sprite_info structures, which are described by the following type definitions :

```
typedef struct {
        UInt16 size;            // bytes for this delta
        Char *ptr;
} delta_info_struct;

typedef struct {
        UInt8   w;              // width of the sprite in pixels (2-64, even)
        UInt8   h;              // height of the sprite in pixels (1-254)
        UInt8   delta_count;    // number of deltas stored for this sprite (0-32)
        UInt8   ws;             // scaling flag ( valid on consoles only )
        UInt16  size;           // bytes per frame, i.e. w x h
        Char   *ptr;
        delta_info_struct delta[MAX_DELTAS];
} sprite_info_struct;
```

There is one sprite_info_struct for each different sprite graphic. Each contains information on a variable

number of deltas, the number being given by **delta_count**. These deltas are small, variable sized, graphic changes which can be applied in any combination to this sprite. Deltas for one sprite cannot be appiled to another sprite.

The **ptr** items in **sprite_info_struct** and **delta_info_struct** are used to hold pointers to the actual position in memory of the graphic for the sprite or delta. These are empty in the file. The numbers are filled in by the game engine when it loads the actual graphics data.

3.2.2 24-bit Remaps

The format used for storing 24-bit remaps is :

```
typedef struct {
        Int16 h, l, s;
} hls_info_struct;
```

This represents a hue/lightness/saturation remap as used in Photoshop. However, if the hue value is greater than 1000 then the remap is loaded from a file instead of calculating it. The filename is remapXXX.tga, where X is hue-1000.

3.2.3 Sprite Graphics

The sprite graphics part of the file contains the actual graphics for the sprites. It stores each sprite followed by all of the deltas for that sprite (if any). The order is the same as the order of **sprite_info_struct** records.

3.2.3.1 Sprites

Sprites are stored in a byte-per-pixel format, in row-major order, i.e. row 0 then row 1 then row 2, etc. They are not compressed at all.

3.2.3.2 Deltas

Deltas have the following format :

```
offset ( 2 bytes )
length ( 1 byte )
data ( length bytes )
```

This is repeated as many times as is necessary to represent the differences between the original sprite and the changed one which the delta is for.
Note that the offset is always relative to the last position used (initially zero).

3.2.3.3 Size Reduction

Sprites are stored in the smallest possible rectangle - the program which generates sprite data must enforce this. If any deltas are larger than the original sprite then the sprite must be stored in a rectangle big enough to hold these deltas and no bigger.

3.2.4 Sprite Numbers

Sprite numbers data is also stored in the style file. This is a list of numbers which is used by the game to reference particular sprite types. The format is :

```
typedef struct {
        UInt16 SPR_ARROW;
        UInt16 SPR_DIGITS;
        UInt16 SPR_BOAT;
        UInt16 SPR_BOX;
        UInt16 SPR_BUS;
```

```
        UInt16 SPR_CAR;
        UInt16 SPR_OBJECT;
        UInt16 SPR_PED;
        UInt16 SPR_SPEEDO;
        UInt16 SPR_TANK;
        UInt16 SPR_TRAFFIC_LIGHTS;
        UInt16 SPR_TRAIN;
        UInt16 SPR_TRDOORS;
        UInt16 SPR_BIKE;
        UInt16 SPR_TRAM;
        UInt16 SPR_WBUS;
        UInt16 SPR_WCAR;
        UInt16 SPR_EX;
        UInt16 SPR_TUMCAR;
        UInt16 SPR_TUMTRUCK;
        UInt16 SPR_FERRY;
} sprite_numbers_struct;
```

Each of these numbers stores the number of sprites of that particular type. The number can be zero if there are no sprites of that type in the style.

3.3 remap_tables (8-bit only)

remap_tables is the area of the style file which stores the colour remap information for the style. It stores 256 seperate remap tables. Each remap table contains 256 bytes. These bytes represent a re-ordering of the original colour palette. To remap a sprite. Each pixel's colour is used as an index into the relevant remap table.
Remaps can be appiled to cars, pedestrians, objects and tile lids. Unused remap tables will store a copy of the original palette.
Tiles can use any remap from 0-255 but sprites (cars, peds, objects) can use only 0-127.

3.4 remap_index

remap_index is a table which stores the table numbers of the 3 remaps which can be applied to each lid tile. The structure is :

UInt8 **remap_index**[256][4];

This an array of 256 sub-arrays, where each sub-array refers to 1 tile. In the sub-array, the elements store remap table numbers. Element 0 always stores 0 (meaning no remap). The other elements can store any number from 0-255.

3.5 palette_index (24-bit only)

Rather than store all of the generated palettes, only unique palettes are stored. A tool called **palcut** takes the original palette data and generates a minimal palette set plus an index, which maps the old palette numbers onto their position in the new set. This index is stored in the 24-bit style file as **palette_index**.

3.6 File Format

The style data, as described above, is stored in a file **styleXXX.grx**. The format of this file is :

name	size	notes
version code	UInt32	currently grx=290, gry=325, g24=336
side_size	UInt32	no. bytes of side_block data
lid_size	UInt32	no. bytes of lid_block data
aux_size	UInt32	no. bytes of aux_block data
anim_size	UInt32	no. bytes of anim data
palette_size	UInt32	no. bytes of palette data (normally 768)

remap_size	UInt32	no. bytes of remap_tables data (normally 65536)
remap_index_size	UInt32	no. bytes of remap_index data (normally 1024)
object_info_size	UInt32	no. bytes of object_info data
car_size	UInt32	no. bytes of car_info data
sprite_info_size	UInt32	no. bytes of sprite_info data
sprite_graphics_size	UInt32	no. bytes of sprite_graphics data
sprite_numbers_size	UInt32	no. bytes of sprite_numbers data
side_block	side_size	
lid_block	lid_size	
aux_block	aux_size	
anim	anim_size	
palette	palette_size	3 bytes per colour RGB
remap_tables	remap_size	
remap_index	remap_index_size	
object_info	object_info_size	
car_info	car_size	
sprite_info	sprite_info_size	
sprite_graphics	sprite_graphics_size	
sprite_numbers	sprite_numbers_size	

3.7 PC Retail Version

The **.grx** file described above is used for the editor only. The 8-bit game loads a **.gry** file, which is the same as the **.grx** with the following exceptions :
- version number is 305
- tile blocks are stored in a paged format (256 x 256 pixel pages)
- extra space may be added at the end of aux_block so that the total number of blocks is a multiple of 4 (the size values stored in the file are not changed, however)
- the sprites are also stored in a paged format - a whole number of pages is always loaded

3.8 24-bit PC Version

The 24-bit PC loads **.g24** files instead of **.gry**. (Note that all other files loaded by the game are the same whether 8-bit or 24-bit).
Differences in the file format are described here.

3.8.1 File Format

The 24-bit style data, as described above, is stored in a file **styleXXX.g24**. The format of this file is :

name	size	notes
version code	UInt32	currently 336
side_size	UInt32	no. bytes of side_block data
lid_size	UInt32	no. bytes of lid_block data
aux_size	UInt32	no. bytes of aux_block data
anim_size	Uint32	no. bytes of anim data
clut_size	Uint32	total bytes of clut data (before paging)
tileclut_size	Uint32	no. bytes of tile cluts
spriteclut_size	Uint32	no. bytes of sprite cluts
newcarclut_size	Uint32	no. bytes of newcar cluts (car remaps + ped remaps)
fontclut_size	Uint32	no. bytes of font cluts

palette_index_size	Uint32	no. bytes of palette index data
object_info_size	Uint32	no. bytes of object_info data
car_size	Uint32	no. bytes of car_info data
sprite_info_size	Uint32	no. bytes of sprite_info data
sprite_graphics_size	Uint32	no. bytes of sprite_graphics data
sprite_numbers_size	Uint32	no. bytes of sprite_numbers data
side_block	side_size	
lid_block	lid_size	
aux_block	aux_size	
anim	anim_size	
clut	clut_size, rounded up to 64K	paged CLUT data : tiles sprites car remaps ped remaps fonts
palette_index	palette_index_size	
object_info	object_info_size	
car_info	car_size	
sprite_info	sprite_info_size	
sprite_graphics	sprite_graphics_size	
sprite_numbers	sprite_numbers_size	

4. Document Revision List

Version	Author	Date	Comments
1.00	K.R. Hamilton	February 20, 1995	Initial version
2.00	K.R. Hamilton	March 10, 1995	Compression
2.10	K.R. Hamilton	March 13, 1995	Flats & pavements
3.00	K.R. Hamilton	March 16, 1995	Synchronous anim
3.10	K.R. Hamilton	March 21, 1995	Palette info
3.20	K.R. Hamilton	March 23, 1995	Word 6
3.30	K.R. Hamilton	March 24, 1995	Column order
4.00	K.R. Hamilton	March 28, 1995	Expanded type map
4.10	K.R.Hamilton	April 13, 1995	Type map revisions
4.20	K.R.Hamilton	April 14, 1995	Type map correction
4.21	K.R.Hamilton	April 27, 1995	Page formatting
4.30	K.R. Hamilton	May 2, 1995	Direction bits
4.40	K.R. Hamilton	May 16, 1995	Style info in map, remove MapCom
4.50	K.R. Hamilton	June 29, 1995	extra byte of type info
5.00	K.R. Hamilton	July 7, 1995	remap bit, sprite info
5.10	K.R. Hamilton	July 14, 1995	flip bits
6.00	K.R. Hamilton	July 26, 1995	object data, CulCom
6.10	K.R. Hamilton	July 27, 1995	add depth to objects
6.20	K.R. Hamilton	August 2, 1995	shallower slopes
6.21	K.R. Hamilton	August 7, 1995	fix map file version
6.22	K.R. Hamilton	August 8, 1995	fix slope numbers
7.00	K.R. Hamilton	August 14, 1995	route information, object rotation, sprite file version code
7.10	K.R. Hamilton	August 24, 1995	animated objects, location data, tidy up

7.20	K.R. Hamilton	August 25, 1995	route types, above types shift
7.30	K.R. Hamilton	September 4, 1995	new object types, add car types info
7.40	K.R. Hamilton	September 27, 1995	sprite reduction
8.00	K.R. Hamilton	October 24, 1995	sep. lid tiles, join style and sprite, bombs, multiple bases
8.10	K.R. Hamilton	November 6, 1995	roadblock data
9.00	K.R. Hamilton	February 5, 1996	mission.ini & language files + nav info & sprite numbers
9.10	K.R. Hamilton	February 9, 1996	column height - back to Uint16 instead of Uint8
9.20	K.R. Hamilton	February 19, 1996	swap status & aux order in object_info & add reserved word in map header
9.30	K.R. Hamilton	March 12, 1996	alignment changes to sprite & object info for console versions
9.40	K.R. Hamilton	March 14, 1996	scaling & more alignment for consoles + car model
10.00	K.R. Hamilton	April 11, 1996	car pos info added to object pos. Remap tables added & remap info in car data
10.10	K.R. Hamilton	April 30, 1996	tile table remapping added
10.15	K.R. Hamilton	April 30, 1996	damage & turning added to car info
10.20	K.R. Hamilton	June 6, 1996	car value added to car info
10.25	K.R. Hamilton	June 6, 1996	car value x $1000
10.30	K.R. Hamilton	July 25, 1996	.gry files
11.00	K.R. Hamilton	October 22, 1996	.g24 files, new car data
11.10	K.R. Hamilton	October 23, 1996	additions to car data
11.20	K.R. Hamilton	December 5, 1996	more addns to car data + pal index
11.21	K.R. Hamilton	December 11, 1996	fix convertible / engine order
12.00	K.R. Hamilton	January 7, 1998	update to match retail version
12.10	K.R. Hamilton	February 3, 1998	loaded_anim structure corrected (with thanks to Paul Ashton)