Introduction

Current version v1.1.2

PAG (Portable Animated Graphics) is a special animation file format designed for high-performance rendering scenarios, which can store vector images, text, bitmaps and sequence frames at a very high compression ratio, and make full use of the hardware acceleration capabilities of each platform to quickly decode and render to screens of various sizes at high speed. The main features of the PAG file format are as follows:

- **Screen rendering.** It is mainly designed for screen rendering scenarios, not for editable animation content exchange between various animation creation tools, so the key direction of optimization is rendering performance and file size.
- **Scalability.** The basic data structure is based on the description of labeled data blocks, which allows for the continuous addition of new action feature support while maintaining backward compatibility with older file formats.
- **High compression rate.** Pure binary data structure, dynamic bitwise storage with extremely high compression rate and similar block centralized compression technologies can achieve an average file size of only 10% to 50% of other formats for the storage of the same animation content.
- **File independence.** A single file can integrate any resources, such as vectors, images, text, sequence frames, and even audio and video. The single-file delivery capability can achieve a more concise workflow.
- **High-speed rendering.** The file format is simple and does not contain any string-matching process. Compared with the decoding speed of text configuration files, binary streams have significant advantages. This format can be optimized during encoding, and the direct data required for the rendered scenario can be converted in advance, so the animation content can be rendered on the screen faster.

PAG files have the suffix.pag

Chapter 1: Basic Data Types

This chapter introduces the basic data types and the more complex data structures formed from them. All other data structures in the PAG document format are composed of these basic types.

Integer and Byte Order

The PAG file format supports 8-bit, 16-bit, 32-bit, 64-bit signed and unsigned integer types. All integer values are stored in the PAG file in binary form. PAG's byte storage uses little-endian byte order: the lowest-order byte is stored at the lowest memory address, and the highest-order byte is stored after it. All integer types are byte pairs. The first bit of an integer value must be stored as the first bit of the byte in the PAG file.

Integer Table

Type	Remarks
Int8	8-bit integer
Int16	16-bit integer
Int32	32-bit integer
Int64	64-bit integer
Uint8	unsigned 8-bit integer
Uint16	unsigned 16-bit integer
Uint32	unsigned 32-bit integer
Uint64	unsigned 64-bit integer

Boolean Type

A byte bit is used in the PAG file to identify the Boolean type.

Boolean Class Type Table

Type	Remarks
Bool	0 : false , 1: true

Float Types

The PAG file supports the single precision float type according to IEEE 754 standard.

Float Type Table

Type	Remarks
Float	Single precision (32 bit) IEEE Standard 754

Array

For the same data type stored continuously, we add the **[n]** symbol after the data type to represent it, where **n** indicates the degree of the array. For example, **Uint8[10]** represents an array of **Uint8** type, with a degree of 10. If two parentheses are added consecutively, such as **Int8 [n][m]**, it means a two-dimensional array, the length of the first dimension array is **m**, and each value type is an **Int8[n]** array.

Encoded Integer

The PAG supports encoded integers of variable byte length. Four encoded integer types are supported.

Encoded Integer Table

Type	Remarks
EncodeInt 32	Variable-length encoded 32 -bit integer
EncodeUint 32	Variable -length encoded 32 -bit unsigned integer
EncodeInt 64	variable-length encoded 64 -bit integer
EncodeUint 64	Variable -length encoded 64 -bit unsigned integer

Variable-length encoding uses bytes as the minimum storage unit, and uses the first seven bits of a byte to store values, and the eighth bit to identify whether there is any value behind. If the 8th bit is 0, it indicates that the value has been read. If the 8th bit is 1, then read one byte down until the length is read (32 bits or 64 bits).

The following is an example of parsing unsigned 32-bit encoded integers:

```
uint32_t ByteBuffer::readEncodedUint32() {
static const uint32_t valueMask = 127;
static const uint8_t hasNext = 128;
uint32_t value = 0;
uint32_t byte = 0;
for (int i = 0; i < 32; i += 7) {
if (_position >= _length) {
Throw(context, "End of file was encountered.");
break;
}
byte = bytes[_position++];
value |= (byte & valueMask) << i;
if ((byte & hasNext) == 0) {
break;
}
}
return value;
}
```

For signed 32-bit encoded integers, the unsigned 32-bit encoded value is read first, and then the high-bit identifier is determined.

The following is the parsing of signed 32-bit encoded integers:

```
int32_t ByteBuffer::readEncodedInt32() {
auto data = readEncodedUint32();
int32_t value = data >> 1;
return (data & 1) > 0 ? -value : value;
 }
```

Bit Type

The bit type value is a variable-length bit field that can represent two types of numbers:

1. Unsigned integer

2. Signed integer

Bit values are not byte aligned. Other types of values, such as Uint8 and Uinit16, are always byte-aligned. If a byte-aligned type immediately follows a Bit type, zero padding is used for the extra bits in the last byte, except for the Bit value.

The following example is a 64-bit data stream. 64 bits represent 9 values of different bit lengths, the last one is the value of Uint16.

The first value of the bit stream is a 6-bit value (BV1), followed by a 5-bit value (BV2) that spans Byte1 and Byte2. The value of BV3 spans Byte2 and Byte3. BV4 is all in Byte3. BV9 is followed by a byte-aligned value, so the remaining 4 bits in the last Byte use zero padding.

Bit Value Type

Type	Remarks
SB[nBits]	Integer (nBits is the number of specified bits)
UB[nBits]	Unsigned integer (nBits is the number of specified bits)

Bit Type Use

The Bit data type generally uses the smallest number of bits required for storage. Most Bit-type fields use a fixed number of bits. Generally, for a group of Bit type data, the minimum number of bits required for storage of this set of data will be calculated. The value of this smallest bit will be stored in another data structure. In this minimum bit range class, the extra bits are filled with zeros in high bits.

Continuous Data Encoding

We use the continuous data encoding method to store a set of continuous data of the same type to achieve the least number of data storage bits.

Consecutive Unsigned Integer Encoding

For the storage of continuous unsigned integers, we add a header area in front of the continuous unsigned integer data. The data in the header area is used to identify the number of bits stored in the continuous unsigned integer data. The following continuous unsigned integer data will be stored according to this number of bits.

Take the storage of continuous 32-bit unsigned integers as an example. The header area is 5 bits (32-bit unsigned integers occupy a maximum of 4 bytes, 32 bits. The range of values that can be represented by 5 bits is 0-31. Removing this situation with 0, the preset addition of 1 can indicate that the value range is 1-32). You should first calculate the maximum value and maximum value of continuous unsigned integer data, and then calculate the number of bits required to store the two values. The maximum value of the two will be stored in the header area, and the continuous unsigned integer data area stores unsigned integer data in sequence according to this value.

Its structure is as follows:

Consecutive Signed Integer Encoding

The storage method of continuous signed integers is the same as that of unsigned integers, the difference is that the nBits data read by unsigned integers all represent numerical content, while the first bit of the nBits bit read from signed integers represents the sign bit (0 positive, 1 negative), and the subsequent consecutive nBits-1 bits represent numerical content.

The process of reading a single signed integer can first follow the unsigned integer and then convert it to a signed integer separately:

```
int32_t ByteBuffer::readBits(uint8_t numBits) {
    auto value = readUBits(numBits);
    value <<= (32 - numBits);
    auto data = static_cast<int32_t>(value);
    return data >> (32 - numBits);
}
```

Consecutive Float Encoding

For continuous float data, we usually retain the original precision and store it continuously according to the IEEE 754 standard without special compression. However, for continuous float data representing some special categories that allow loss of precision, the encoding

method is to convert the float lossy data into integer data through the **float/precision**, and then follow the above method for continuous integer data format to encode. Currently, the following types of float data in the PAG file can be converted to integer storage according to the corresponding precision:

Type	Precision	Remarks
SPATIAL_PRECISION	0.05	When the float number represents coordinate points in space
BEZIER_PRECISION	0.005	When the float number represents the precision of the Bezier curve easing parameter
GRADIENT_PRECISION	0.00002	When the float number represents the precision of the gradient interpolation position parameter

Specific to the decoding process, the parsed integer data is multiplied by the precision to obtain specific float data.

Continuous Boolean Type Encoding

In the computer storage method, Bool type data occupies 1 byte, 8 bits, but the effective data bit is only 1 bit, and redundant storage data occupies 7 bits. For continuous Bool type data, we only use one bit to identify, thereby reducing redundant data storage.

Time

Time in the PAG file is uniformly described using Int64. In order to improve the efficiency of caching during rendering, the smallest time unit is 1 frame, and the number of frames divided by the frame rate can be converted to an external time in seconds. But when storing files, all time values are stored as EncodeUint64 instead of EncodeInt64. When describing animation effects, the time in most cases is a positive number, and the negative time only appears during the rendering calculation process. However, using unsigned integer storage can occupy 1 bit less space than signed ones, so unsigned integers are uniformly used when storing time to files. In addition, even if there is a small probability of encountering negative time, the negative time can be restored normally according to unsigned storage and reading. The difference is that in the case of a negative number, converting it to an unsigned integer is a huge number, which may take up an extra byte of space for encoding integers. However, the probability of storing negative time in files is very low in general.

ID

The ID in the PAG file is uniformly described by Uint32, and EncodeUint32 is used for storage. Usually, Composition, Layer, and Mask will have ID class attributes.

Enum

Uint8 is used in PAG files to store enumerated types. For specific enumeration types, refer to Chapter 2: Enumeration.

String

String types use a null character to mark the end.

Stirring Types

Field	Type	Remarks
String	Uint8[0~n]	non-empty character array
StringEnd	Uint8	marks the end, always 0

Point

Point is used to record the position of the x and y axes.

Point Types

Field	Type	Remarks
x	Float	x-axis coordinate
y	Float	y-axis coordinate

Ratio

Ratio is used to store the ratio.

Ratio Type

Field	Type	Remarks
numerator	EncodedInt32	numerator
denominator	EncodedUint32	denominator

Color

Color represents a color value, usually composed of 24-bit red, green, and blue colors.

Color Types

Field	Type	Remarks
Red	Uint8	Red value (0 ~ 255)
Green	Uint8	Green value (0 ~ 255)
Blue	Uint8	Blue value (0 ~ 255)

FontData

FontData identifies the font.

FontData Types

Field	Type	Remarks
fontFamily	String	Font description
fontStyle	String	Font style

AlphaStop

AlphaStop describes the gradient information of transparency.

AlphaStop Types

Field	Type	Remarks
position	Uint16	Start point. The float type is stored in Uint16, and the real value needs to be multiplied by The precision GRADIENT_PRECISION, Uint16() * 0.00002f
midpoint	Uint16	Middle point. The float type is stored in Uint16, and the real value needs to be multiplied by the precision GRADIENT_PRECISION, Uint16() * 0.00002f
opacity	Uint8	Transparency (0 ~ 255)

ColorStop

ColorStop describes the gradient information of color.

AlphaStopTypes

Field	Type	Remarks
position	Uint16	Start point. The float type is stored in Uint16, and the real value needs to be multiplied by the precision GRADIENT_PRECISION, Uint16() * 0.00002f
midpoint	Uint16	Middle point. The float type is stored in Uint16, and the real value needs to be multiplied by the precision GRADIENT_PRECISION, Uint16() * 0.00002f
color	Color	Color value

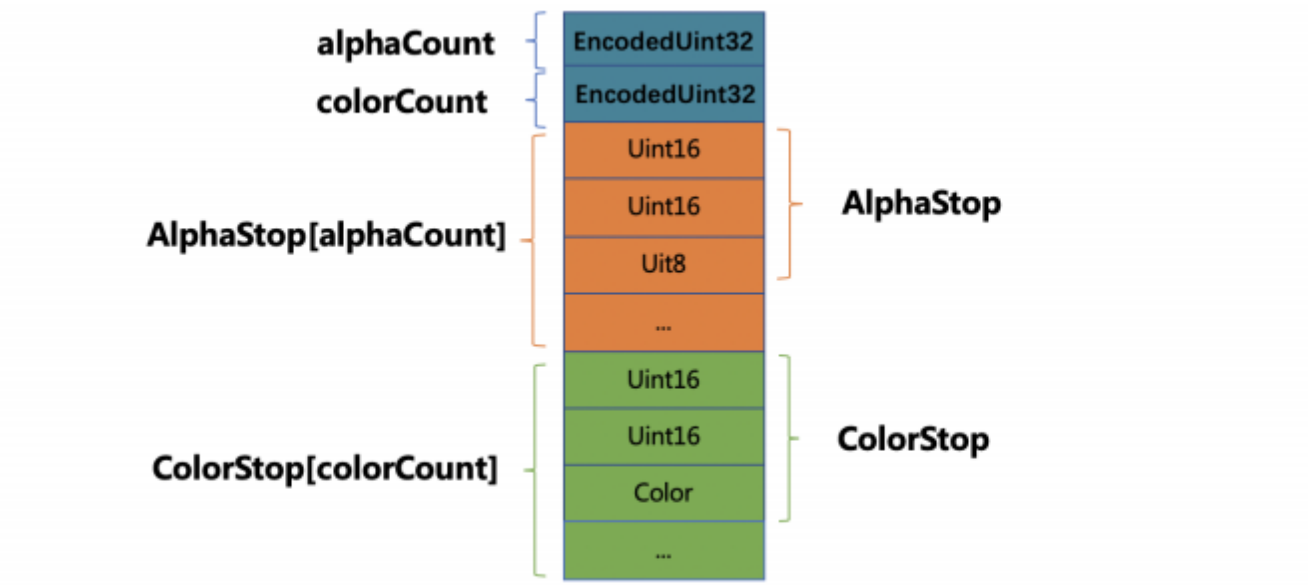
GradientColor

GradientColor is used to describe the gradient information of color and transparency.

GradientColor Types

Field	Type	Remarks
alphaCount	EncodedUint 32	The length of the transparency gradient information array. A transparency gradient information will include, starting point: position, middle point: midpoint, and transparency value opacity
colorCount	EncodedUint 32	The color gradient information array. A color gradient information contains, starting point: position, middle point : midpoint, and color value color
alphaStopList	AlphaStop[alphaCount]	Consecutive alphaCount AlphaStops
colorStopList	ColorStop[colorCount]	Consecutive colorCount ColorStops

The corresponding storage structure is as follows:



Text Document

TextDocument text information includes text, font, size, color and other basic information.

TextDocument Types

Field	Field Type	Remarks
applyFillFlag	UB [1]	Whether to apply a fill flag
applyStrokeFlag	UB [1]	Whether to apply a stroke flag
boxTextFlag	UB [1]	
fauxBoldFlag	UB [1]	Whether to apply a bold flag
fauxItalicFlag	UB [1]	Whether to apply an italic flag
strokeOverFillFlag	UB [1]	
baselineShiftFlag	UB [1]	
fi rstBaseLineFlag	UB [1]	
boxTextPosFlag	UB [1]	
boxTextSizeFlag	UB [1]	
fi llColorFlag	UB [1]	Whether there is fill color information flag
fontSizeFlag	UB [1]	Whether there is a font size flag
strokeColorFlag	UB [1]	Whether there is a stroke color flag
strokeWidthFlag	UB [1]	Whether there is a stroke width flag
textFlag	UB [1]	Whether there is a text flag
justifi cationFlag	UB [1]	Whether there is alignment information flag

leadingFlag	UB [1]	
trackkingFlag	UB [1]	
hasFontDataFlag _	UB [1]	Whether to include font information
	UB [5]	All 0, byte alignment
baselineShift	Float	if baselineShiftFlag == 1
firstBaseLine	Float	if first BaseLineFlag == 1
boxTextPosFlag	Point	if boxTextPosFlag == 1
boxTextSizeFlag	Point	if boxTextSizeFlag == 1
fillColor	Color	if fillColorFlag == 1
fontSize	Float	if fontSizeFlag == 1
strokeColor	Color	if strokeColorFlag == 1
strokeWidth	Float	if strokeWidthFlag == 1
text	String	if textFlag == 1
justification Flag	Uint8	if justificationFlag == 1
leadingFlag	Float	if leadingFlag == 1
trackingFlag	Float	if trackingFlag == 1
fontID	EncodedUint32	if hasFontDataFlag == 1

Path

Path is used to identify information such as a path. The main information includes an action list and a coordinate list.

PathVerb Types

Field	Type	Attribute Value	Coordinate Data	Remarks
Close	UB [3]	0	No need	Close the current path to the path start point
Move	UB [3]	1	**Point**	Move the coordinate point to the specified position**,** and **Point** indicates the target point to move to
Line	UB [3]	2	**Point**	Draw a line, and **Point** indicates the target point to draw a straight line to
HLine	UB[3]	3	**Float**	Draw a horizontal line, with the X-axis moving while the Y-axis remains the same as the previous value. The **float** data represents the target position of X- axis movement.
VLine	UB[3]	4	**Float**	Draw a vertical line, with the Y-axis moving and the X-axis remaining the same as the previous value. The **float** data represents the target position of the Y-axis movement
Curve01	UB[3]	5	**Point**, **Point**	Draw a cubic Bezier curve, with the first control point having the same value as the end point of the previous action. The two **points** represent the second control point and the end point, respectively.

Curve10	UB[3]	6	**Point**, **Point**	Draw a cubic Bezier curve, with two **points** representing the first and second control points, respectively, and the ending point being the same as the second control point.
Curve11	UB[3]	7	**Point**, **Point**, **Point**	Draw a cubic Bezier curve with three **points** representing the first control point, the second control point and the end point in sequence.

Path Types

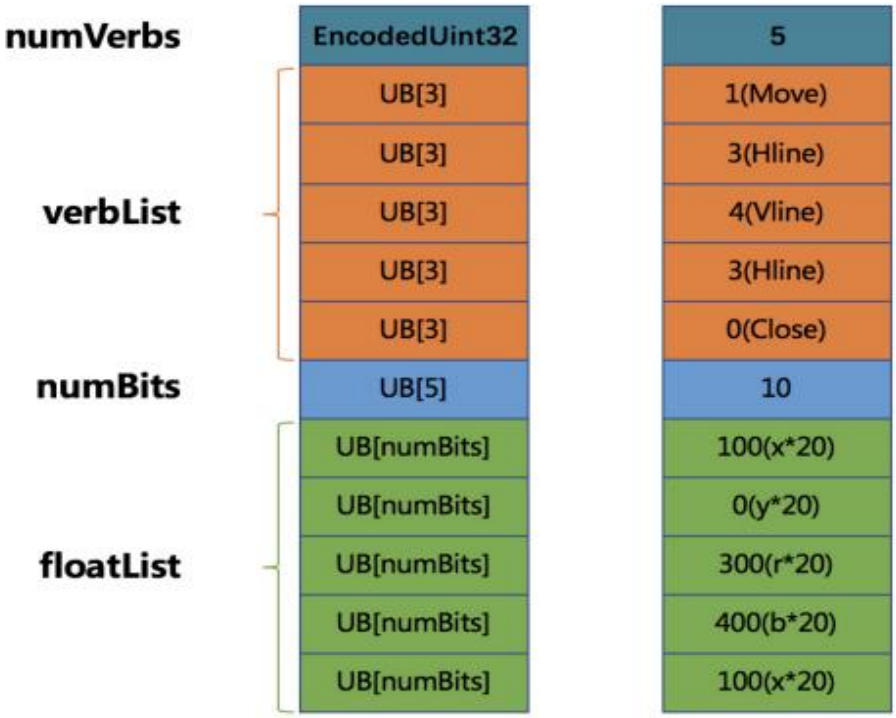
Field	**Type**	**Remarks**
numVerbs	EncodedUint32	The length of the action list
verbList	UB[3][numVerbs]	An array of action lists with a length of numVerbs, each value of the array is identified by a UB[3] to an enumeration value in PathVerb.
numBits	UB[5]	Indicates the number of bits occupied by each value in the next floatList
floatList	SB[numBits] [floatNum]	Used to provide an array of float values for coordinate points, the length is floatNum, and each value of the array is stored by a SB[numBits]

The calculation of **floatNum** can refer to the **coordinate data** column in the PathVerb table, where a **Point** can be equivalent to two consecutive **Floats**, representing the x and y coordinate values of the **Point** respectively. The value of **floatNum** is the total number of required **Float** data accumulated based on the number of **Floats** required for each action type in the VerbList. For each value in the floatList array, you need to read an **SB[numBits]** value first, then multiply it by SPATIAL_PRECISION to get a **Float** value.

For example, a rectangle **(x: 5, y: 0, r: 15, b: 20)** can be described by a Path data structure as follows:

- Execute Move to point (5, 0), need to record two Float values: 5, 0
- Execute HLine to point (15, 0), only need to record one Float value: 15
- Execute VLine to point (15, 20), only need to record one Float value: 20
- Execute HLine to point (5, 20), only need to record one Float value: 5
- Execute Close to close the rectangle and return to the starting point (5, 0), without recording any Float value.

A total of 5 actions (Move, HLine, VLine, HLine, Close) and 5 Float coordinate data (5, 0, 15, 20, 5) need to be recorded. The corresponding Path storage structure is as follows:



The storage of each Float value in **floatList** is first multiplied by **20 (1/SPATIAL_PRECISION)**, converted to an integer, and then stored as **SB[numBits]**. Among them, **numBits** is calculated based on the maximum value of **400** stored in the coordinate data. The binary representation of **400** is **110010000**, occupying 9 bits. After adding the sign bits, at least 10 bits are needed. Finally, when the length of numBits is 10, it is sufficient enough to include each data in floatList.

Byte Stream ByteData

ByteData identifies the length and content of the byte stream.

ByteData Types

Field	Type	Remarks
length	EncodedUint32	Byte length
data	Byte[length]	Read length bytes

BitmapRect

Bitmap information.

BitmapRect Types

Field	Field Type	Remarks
x	EncodedInt32	
y	EncodedInt32	
fileBytes	ByteData	Image data

VideoFrame

Video frame information.

VideoFrame Types

Field	Field Type	Remarks
frame	Time	
fileBytes	ByteData	Video frame data , the byte stream does not contain (0, 0, 0, 1) four-byte Start Code

Chapter 2: Enumeration

This chapter mainly describes the enumeration types used in PAG files and their meanings.

BlendMode

BlendMode Types

Type	Value	Remarks
Normal	0	
Multiply	1	
Screen	2	
Overlay	3	
Darken	4	
Lighten	5	
ColorDodge	6	
ColorBurn	7	
HardLight	8	
SoftLight	9	
Difference	10	
Exclusion	11	
Hue	12	
Saturation	13	
Color	1 4	
Lu minosity	1 5	
Add	1 6	
DestinationIn	2 1	
DestinationOut	2 2	
DestinationATop	2 3	
SourceIn	2 4	
SourceOut	2 5	
Xor	2 6	

TrackMatteType

Type	Value	Remarks
None	0	
Alpha	1	
AlphaInverted	2	
Luma	3	
LumaInverted	4	

MaskMode

Coverage type for the mask

MaskMode Types

Type	Value	Remarks
None	0	
Add	1	
Subtract	2	
Intersection	3	
Lighten	4	
Darken	5	
Difference	6	
Accum	7	

PolyStarType

PolyStarType Types

Type	Value	Remarks
Star	0	
Polygon	1	

CompositeOrder

Composite Order Types

Type	Value	Remarks
BelowPreviousInSameGroup	0	
AbovePreviousInSameGroup	1	

FillRule

FillRule Types

Type	Value	Remarks
NonZeroWinding	0	
EvenOdd	1	

LineCap

Type	Value	Remarks
Butt	0	
Round	1	
Square	2	

LineJoin

LineJoin Types

Type	Value	Remarks
Miter	0	
Round	1	
Bevel	2	

GradientFillType

GradientFillType Types

Type	Value	Remarks
Linear	0	
Radial	1	

MergePathsMode

MergePathsMode Types

Type	Value	Remarks
Merge	0	
Add	1	
Subtract	2	
Intersection	3	
ExcludeIntersections	4	

TrimPathsType

TrimPathsType Types

Type	Value	Remarks
Simultaneously	0	
Individually	1	

ParagraphJustification

ParagraphJustification Types

Type	Value	Remarks
LeftJustify	0	
CenterJustify	1	
RightJustify	2	
FullJustifyLastLineLeft	3	
FullJustifyLastLineRight	4	
FullJustifyLastLineCenter	5	
FullJustifyLastLineFull	6	

Chapter 3: The overall structure of PAG

This chapter mainly introduces the structure of the PAG file and an overview of each element.

PAG File Structure

PAG files are mainly composed of FileHeader and TagBlock. TagBlock represents a data block consisting of a Tag list. All tags have the same structure, so if you encounter something you don't understand when parsing a PAG file, you can skip the current tag directly.

File Header

All PAG files have the following structure at the beginning of the file. For the type field, please refer to the definition in Chapter 1.

FileHeader Structure

Field	Type	Remarks
Signature	Uint8	signature byte, always ' P '
Signature	Uint8	signature byte, always ' A '
Signature	Uint8	signature byte, always ' G '
Version number	Uint8	File version number, for example: 0x04 means PAG version 4
File length	Uint32	File length (whole file, including FileHeader length)

Compression method	Int8	The compression method of the identification file, reserved

The PAG file headers all start with the three characters ' P ' ' A ' ' G '.

Then the version number of the file is recorded. Note that the version number of the file is a value, not a character. For example, version 4 stores Ox04 instead of ASCII characters ' 4' (Ox 34). The current file version number is **1**.

The file length is the total length of the PAG file, including the FileHeader section.

The compression method records the internal compression technique of the PAG file, which has not been used yet and is reserved.

TagBlock Structure

Tag Block represents a data block consisting of Tag lists.

The structure of TagBlock is shown as follows:

Among them, TagEnd is a special Tag, which is usually used to mark the end of the loop at this level and there is no more Tag structure to read. Some specific Tags will internally define whether they contain other sub-Tag lists, and also use TagEnd at the end to mark the end of the sub-Tag list loop and there are no more Tags to read.

Tag Structure

A Tag consists of two main parts: **TagHeader** and **TagBody**.

The structure of Tag is as follows:

TagHeader

TagHeader records TagCode (type ID of Tag) and the byte stream length of TagBody. Since the TagBody may be very short or very long, for the sake of compression ratio, we divide the TagHeader into two types of short and long formats for storage. The short type TagHeader is used to record the maximum 62 bytes of TagBody data. The long type TagHeader uses a 32-bit unsigned integer to record the length of TagBody, which can store up to 4GB of data.

TagHeader (short) Types

Field	**Type**	**Remarks**
TagCodeAndLength	Uint16	The first 10 bits are the TagCode and the last 6 bits are the length of the TagBody.

Note: The TagCodeAndLength field reads two bytes at a time, instead of 10 bits followed by 6 bits. PAG files are stored in little-endian byte order, so the two storage methods are different.

If TagBody is 63 bytes or longer, it is stored in a long type TagHeader. The long type TagHeader contains a short type TagHeader structure, where the length of the TagBody recorded by the short type TagHeader is a fixed value: 0x3f (63), followed by a 32-byte unsigned integer representing the true length of TagBody.

TagHeader (long) Types

Field	Type	Remarks
TagCodeAndLength	Uint16	The first 10 bits are used as TaCode, and the last 6 bits are fixed as 0x3f to mark the TagHeader as long type.
Length	Uint32	The actual length of TagBody

TagBody

TagBody only defines a byte stream block, and the specific rules for parsing content are defined according to different TagCode categories. TagBody is also allowed to be absent. For example, the length of the TagBody read from its TagHeader is 0 for the special TagEnd mentioned earlier. Since the byte stream inside the TagBody can be completely customized, it can also be defined to continue to include a list of sub-Tags, that is, there is a case where a Tag can also contain one or more sub-Tags, so as to realize nesting. The following chapters will explain in detail how the TagBody corresponding to different TagCode categories is stored. Generally speaking, the internal storage of TagBody is the specific data of an attribute list. The storage methods can be divided into two categories:

- If the attribute list described in TagBody is fixed from type to quantity, then the internal structure of the TagBody will directly arrange and combine the basic data types provided in Chapter 1 for definition. Similar to the definition method of the **Path** data structure, a new data structure is formed by using the existing data structure arrangement. When decoding, just pass the TagBody directly to the decoding module of the corresponding category. For this type of Tag, we will directly list the data structure of its TagBody in a table in the following chapters.
- If the attribute list described in the TagBody is uncertain from type to quantity, a large number of additional identifier fields will be required. At this time, using the above-mentioned method to define the data structure will waste a lot of file space. For this type of Tag, we will use the dynamic data structure **AttributeBlock** to describe the content of the entire TagBody. Later chapters will introduce **AttributeBlock** in detail.

Chapter 4: AttributeBlock

The previous chapters introduced the data structure of Tags. Before explaining how the TagBody corresponding to each Tag type is stored, we first introduce a new dynamic data structure: **AttributeBlock**. When the attribute list to be stored is uncertain from type to quantity, a large number of additional status identifier fields will be required. This dynamic data structure is mainly used to maximize the compression of these identifiers and dynamically match different decoding modules according to the identifiers.

Value and Property Attributes

AttributeBlock is usually used to describe how to store the data of a set of attribute lists. Here we take the data structure of the Mask as an example. The list of attributes it needs to store is as follows:

Field	Type	Remarks
id	Uint32	Mask ID
inverted	Bool	Indicates whether the mask is inverted
maskMode	Uint8	The blend mode of the mask
maskPath	Property <Path>	The vector path of the mask
maskOpacity	Property<Uint8>	Transparency of the mask
maskExpansion	Property<Float>	Edge extension parameters of the mask

So far, all we have introduced are basic data types. You can see that in the attribute list of Mask, the types of the first three attributes are also this data type. These attributes are characterized by containing only one data value, and I can collectively refer to these attributes as Value attributes. Here we introduce another type called timeline attribute. You can use **Property<T>** to refer to any kind of timeline attribute, and T can be any basic data type such as Bool, String, Int8, Point, Path... and so on. If we replace **T** with a specific type, such as **Property<Point>**, then it represents the time attribute of the **Point** class.

The main difference between the Value attribute and the Property attribute is that the Value attribute only contains one data value, while the Property attribute usually contains multiple data values of the entire timeline. Depending on the number of keyframes, it can contain one or more key points' data values. It can be simply understood as a collection of a series of Property values on the timeline. The Property attribute also contains time easing parameters and spatial easing parameters, which are used to control how the data values between keyframes produce instantaneous interpolation. The following chapters will introduce the storage structure of the Property attribute in detail. Here we only need to understand its timeline concept.

For multiple attributes of Mask, the easiest way to store is to store the complete structure of each type, but obviously, this is a waste of space. We can find many situations where redundant data exists:

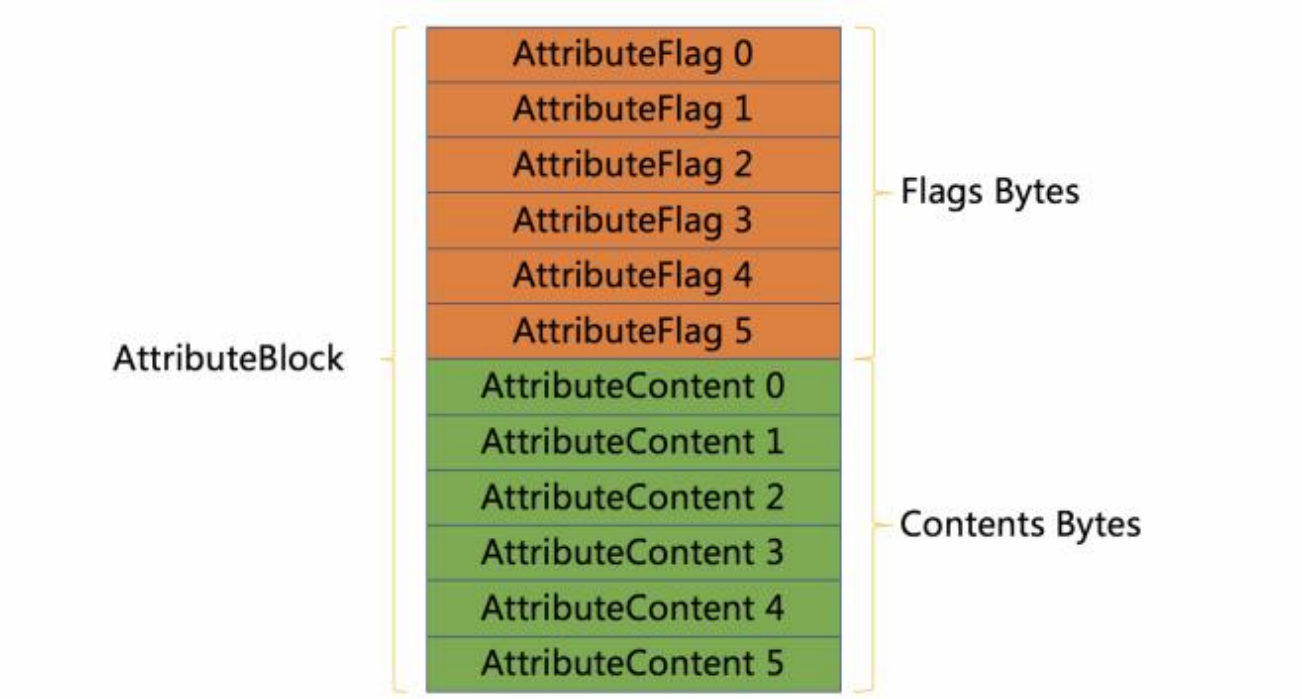
- The inverted attribute is a Bool value, which needs to occupy 1 byte, but actually, only 1 bit can be used for identification.
- maskMode is usually equal to the default value. In this case, there is no need to store the actual value, and 1 bit is used directly to indicate that it is equal to the default value.
- All Property attributes: 1. If there is no spatial easing parameter, the relevant data may not be stored. 2. If there is no keyframe, that is, there is only one data value, the storage structure of the entire keyframe can be saved, and only one value can be stored. 3. If this value is still equal to the default value, you can only use 1 bit to indicate that it is equal to the default value.

According to the above characteristics, it can be found that each attribute may have many special status flags, and making full use of these status flags can significantly reduce the file size in most cases. Especially for the Property attribute, a very complex data structure is required to record the complete storage of a timeline attribute. However, in most cases, the Property attribute may not have a keyframe, that is, it can be degenerated into a Value attribute for storage, so the unnecessary space occupied can be significantly reduced. Therefore, we use the data structure of **AttributeBlock** to describe this storage scenario that requires a large number of status flags.

AttributeBlock Structure

AttributeBlock mainly consists of two parts : by **AttributeFlag** The flag field consisting of lists and **the AttributeContent** A content area composed of lists. Tool The body structure is as follows:

AttributeBlock is mainly composed of two parts: the flag bit area composed of the **AttributeFlag** list and the content area composed of the **AttributeContent** list. The specific structure is as follows:



The sequence of each **AttributeFlag** corresponds to the sequence of **AttributeContent**. Each time an attribute list is given, we split the flag bit information and data content information of each attribute into a pair of **AttributeFlag** and **AttributeContent** structures, and then store them in order. For example, **AttributeFlag 0** and **AttributeContent 0** correspond to the **ID** attribute of the first item in the Mask attribute list. It can be seen from the figure that the storage structure is to store all the Flag lists before starting to store all the Content lists. This is done because each item in the Flag area is usually represented by a bit, and centralized storage can avoid frequent byte alignments, resulting in a lot of space waste. After reading the Flag area, a unified byte alignment will be performed, and the extra bits generated by the byte alignment will be set to 0, and then continue to store the **AttributeContent** area from the integer byte position.

Note: In order to save file space, the number of attribute lists is not written into the file, but is hardcoded in the parsing code because the number of attributes for each specific AttributeBlock is fixed. During the parsing process, the number of attributes to be parsed can be known in advance.

AttributeFlag

By summarizing the characteristics of the Value attribute and Property attribute, we abstract the following **AttributeFlag** data structure.

AttributeFlag Types

Field	**Type**	**Remarks**
exist	UB[1]	Identifies whether the corresponding AttributeContent exists
animatable	UB[1]	Whether there is keyframe information, this field exists only when exist is 1
hasSpatial	UB[1]	Whether there is spatial easing information, this field exists only when both exist and animatable are 1

Each attribute can parse out such a data object describing the flag information, which is used to assist in decoding the subsequent **AttributeContent** area. However, the specific storage length of **AttributeFlag** generated by each attribute is dynamic, and the value range is **0 to 3 bits**. For example, the Value attribute may only occupy 1 bit at most, and only the exist flag will be read during decoding. The Property attribute will use up to 3 bits to store flag information. For example, when a Property attribute does not contain keyframes and the attribute value is equal to the default value, it only uses 1 bit (0) to store **AttributeContent**. If the exist flag is read as 0 during decoding, subsequent reads of the flag will be discarded. In the limit case, the entire attribute list does not contain keyframes and is equal to the default value, and finally only occupies the number of bits in the list to store all the exist flags as 0, and the entire Content area is empty. This can significantly reduce the file size that needs to be logged.

AttributeType

As mentioned earlier, the actual storage size of **AttributeFlag** may be 0 to 3 bits, and the Value type attribute will only read 1 bit fixedly. Therefore, when decoding, it is also necessary to know the specific attribute type in order to determine the reading rules of **AttributeFlag**. Earlier, we roughly divided attributes into Value and Property categories. In fact, they can be subdivided into multiple subcategories to further compress the space according to attribute characteristics. The following are all attribute categories used in the PAG file, and their corresponding bits in the Flag area:

AttributeType Types

Type	Flag Area	Remarks
Value	Fixed to 1 bit	Ordinary Value attribute
FixedValue	Not occupied	Fixed Value attribute
BitFlag	Fixed to 1 bit	Value attribute of Bool value type
SimpleProperty	Occupy 1 ~ 2 bits	Simple animation property
DiscreteProperty	Occupy 1 ~ 2 bits	Discrete animation property (no interpolation)
MultiDimensionProperty	Occupy 1 ~ 2 bits	Multi-dimensional time easing animation property
SpatialProperty	Occupy 1 ~ 3 bits	Spatial easing animation property
Custom	Fixed to 1 bit	Extension type, custom data reading rules

The example code for reading AttributeFlag based on different AttributeTypes is as follows:

```
AttributeFlag ReadAttributeFlag( ByteBuffer* byteArray, const AttributeBase* config) {
AttributeFlag flag = {};
auto attributeType = config->attributeType;
if (attributeType == AttributeType::FixedValue) {
flag.exist = true;
return flag;
}
flag.exist = byteArray->readBitBoolean();
if (!flag.exist ||
attributeType == AttributeType::Value ||
attributeType == AttributeType::BitFlag ||
attributeType == AttributeType::Custom) {
return flag;
}
flag.animatable = byteArray->readBitBoolean();
if (!flag.animatable || attributeType != AttributeType::SpatialProperty) {
return flag;
}
flag.hasSpatial = byteArray->readBitBoolean();
```

```
return flag;
}
```

It can be seen that the FixedValue category does not read data from the Flag area, but directly returns the case where exist is true, indicating that the data identifying the attributes of this category will always exist.

Note: In order to save file space, AttributeType is not written into the file, but is hardcoded and configured in the parsing code. As the AttributeType of each attribute is not likely to change, the specific attribute type to be parsed can be known in advance during the parsing process.

AttributeContent

According to the previous steps, we have been able to decode the AttributeFlag object, and then combined with the known AttributeType, we can start decoding the corresponding data for each AttributeContent. The following are the parsing rules corresponding to each AttributeType:

- Value: If exist is true, read the AttributeContent data value, otherwise use the default value.
- FixedValue: Ignore all flags and directly read the AttributeContent data value.
- BitFlag: directly use the value of exist to return as Bool data, without reading AttributeContent data.
- SimpleProperty, DiscreteProperty, MultiDimensionProperty, SpatialProperty: If animatable is false, directly follow the reading rules of the AttributeType Value, and judge whether to read a data value from AttributeContent or use the default value according to exist. If animatable is true, follow the decoding process of Property. The different Property types of these subdivisions and the hasSpatial flag are only used in decoding the internal data structure of the Property. The decoding rules inside Property will be described in detail later.
- Custom: All the previous categories can use the general decoding module, but some AttributeContent may not be an attribute. After configuring the type as Custom, each Tag can specifically define which module this AttributeContent should use for parsing. The exist flag is used to determine whether there is a corresponding AttributeContent. In this way, custom data blocks can be inserted into the attribute list for overall storage.

Note: In order to save file space, the default value of each attribute is not written into the file, but is hardcoded and configured in the parsing code. Because the default value of each attribute is not likely to change, the default value of the attribute to be parsed can be known in advance during the parsing process.

After the above analysis, we have been able to completely decode the data structure of AttributeBlock. Given any set of attribute lists, you only need to list the necessary information such as the attribute order, the basic data type of each attribute, the attribute type, and the default value, and then refer to the previous AttributeBlock rules for decoding. If two columns of Attribute Type and Default Value appear in the subsequent data structure table, it means that it needs to be parsed according to the AttributeBlock data structure. The AttributeBlock structure table corresponding to the previous Mask can be defined as follows:

AttributeBlock Structure Table for Mask

Field	Data Type	Attribute Type	Defaults	Remarks

id	EncodedUint32	FixedValue	0	Mask ID
inverted	Bool	BitFlag	false	Indicates whether the mask is inverted
maskMode	Uint8	Value	1	Blend mode of the mask
maskPath	Path	SimpleProperty	Empty Path object	Vector path of the mask
maskOpacity	Uint8	SimpleProperty	255	Transparency of the mask
maskExpansion	Float	SimpleProperty	0	Edge extension parameters of the mask

Chapter 5: Property

Property is the basic unit of dynamic properties, and the properties on most objects are Property. Because it is widely used, optimizing and compressing the storage structure of a single Property can significantly reduce the size of the entire file. Property is generally not stored separately but is stored as an AttributeContent in the AttributeBlock dynamic data structure alongside other Property or Value attributes as an attribute list. For the analysis of AttributeBlock, please refer to the previous chapter. Therefore, according to the description in the previous chapters, when we parse to the AttributeContent area corresponding to the Property attribute, we can then access the configuration parameters such as **Data Type**, **AttributeType, and Default Value**, as well as the previously read **AttributeFlag** data object. Through these auxiliary parameters, we will start to decode the content of the Property.

Property Structure

The **Property** structure is mainly composed of the keyframe list of the **keyframe** structure, and the length of the list can be 0 to more. When the length of the keyframe is 0 (that is, AttributeFlag.animatable is false), the entire Property attribute contains only one valid value, so it can be degenerated into the Value attribute for storage, and continue to judge whether to read a data value from AttributeContent or set it as the default value. This part of the reading rules has been described before, and the content of this chapter continues to describe how to parse the AttributeContent corresponding to a Property attribute when AttributeFlag.animatable is true. Parsing the structure of a Property is actually parsing the structure of a set of keyframe lists.

Keyframe Data Structure

Property usually contains several keyframe information. A key frame includes the start and end time of the frame, as well as the start and end attribute values, as well as the type of differentiator that identifies the calculation method for attribute values, time easing parameters, etc. For complex Property attributes, its keyframes may also contain multi-dimensional time easing parameters or additionally contain spatial easing parameters. Let's first look at the data fields that a **keyframe** needs to record:

KeyFrame Data Lists

Field	**Type**	**Remarks**
startValue	Generic value (any basic data type)	Start value
endValue	Generic value (any basic data type)	End value
startTime	Time (Int64)	The start time value of the keyframe
endTime	Time (Int64)	The end time value of the keyframe
interpolationType	Enum (Uint8)	Interpolation type
bezierOut	Point[dimensionality]	Time easing parameter array (the first control point of the Bezier curve)
bezierIn	Point [dimensionality]	Time easing parameter array (the second control point of the Bezier curve)
spatialOut	Point	Spatial easing parameters (the first control point of the Bezier curve)

spatialIn	Point	Spatial easing parameters (the second control point of the Bezier curve)

StartValue and endValue represent the start value and end value of this keyframe interval, and the corresponding startTime and endTime represent the start time and end time of this keyframe interval. Therefore, when the value is equal to startTime, startValue will be returned; when the value is equal to endTime, endValue will be returned. At the moment between startTime and endTime, the value is determined by the interpolationType. The interpolation types are as follows::

KeyframeInterpolationType

Type	**Value**	**Remarks**
None	0	Invalid
Linear	1	Linear interpolation
Bezier	2	Use Bezier curve interpolation based on time easing parameters
Hold	3	The entire interval is equal to startValue except for the endTime moment, and endTime can switch to endValue instantly.

Keyframe Compression Method

Combining all the above data structures, it can be concluded that not all data fields of the keyframe need to be stored completely. There are mainly the following scenarios to save storage space:

- When the interpolation type is not equal to **Bezier**, there is no need to store the time easing parameters bezierOut and bezierIn.
- When the property type is **DiscreteProperty**, it means that the interpolation type can only be **Hold**, and there is no need to store interpolationType. This situation usually occurs when the underlying data type of an attribute is a Bool value or an enumeration. Since its data is discrete, the intermediate interpolation between true and false is essentially impossible.
- When the property type is MultiDimensionProperty, it means that the time easing parameter is composed of multiple **Bezier** curves, each individually controlling the subeasing of the data value, usually appearing on the timeline attribute representing scaling. The specific number of Bezier curves is determined according to the data types of startValue and endValue. For example, when the data type is Point, the time easing parameter is a 2-dimensional array, and two Bezier curves control the independent easing of the x and y axes respectively. For cases where the dimension is not a MultiDimensionProperty, we do not need to determine the dimension based on the underlying data type. By default, only the one-dimensional time easing parameters are stored.
- When the attribute type is **SpatialProperty**, it means that the key frame may have spatial easing parameters. At this time, it is necessary to judge whether the actual current keyframe has these parameters according to the AttributeFlag.hasSpatial flag. Only this attribute needs to use the third hasSpaital flag on AttributeFlag. For other attribute types, there is no need to judge or store the spatial easing parameters spatialOut and spatialIn.

In addition to saving storage space through judgment in the above scenarios, we also adopt other compression strategies: when actually storing the keyframe list, we store one type of data for each keyframe in sequence and then store the next type of data collectively instead of storing a keyframe and then storing the next keyframe. The advantage of this is that similar data can be compressed centrally. For example, interpolationType usually only occupies 2 bits, and centralized storage can reduce the extra space waste caused by byte alignment. For another example, since the startValue and startTime of the next keyframe are always equal to the endValue and endTime of the previous keyframe, centralized storage can also skip duplicate data between frames that do not need to be stored.

Property Storage Structure

The specific storage structure of the Property is as follows:

Note: This only describes the read storage structure when AttributeFlag.animatable is true. If AttributeFlag.animatable is false, refer to the previous method to judge whether to read a data value from AttributeContent or use the default value.

Property Types

Field	Type	Remarks
numFrames	EncodedUint32	The length of the keyframe array.
interpolationTypeList	UB[2][numFrames]	The interpolation type corresponds to each keyframe. It read numFrames times in total. Skip this block if the property type is **DiscreteProperty**.
timeList	EncodedUint64[numFrames +1]	The start and end time of each Keyframe. Since the start time of the next keyframe is equal to the end time of the previous keyframe, only read numFrames+1 times.
valueList _	PropertyValueList	The start value and end value of each Keyframe. Different basic data types have different storage methods, and the reading rules of PropertyValueList will be introduced in detail later.
timeEaseNumBits	UB[5]	The number of storage bits occupied by each time easing parameter component.
timeEaseList	TimeEaseValueList	An array of time easing parameters. The reading rules of TimeEaseValueList will be introduced in detail later.
spatialFlagList	UB[numFrames * 2]	An array of flags, indicating whether each subsequent keyframe contains spatialIn and spatialOut parameters respectively.
spatialEaseNumBits	UB[5]	The number of storage bits occupied by each spatial easing parameter component.
spatialEaseList	SpatialEaseValueList	Spatial easing parameter array. The reading rules of SpatialEaseValueList will be introduced in detail later. If the property type is not **SpatialProperty** or **AttributeFlag.hasSpatial** is false, skip this block.

Note: A byte alignment is performed at the end of each block read. It will skip the remaining bits that have not been read, start from the next integer byte position, and then read the next block.

PropertyValueList

The PropertyValue block stores a list of basic data types. The content of the list represents the start value and end value of each keyframe. Since the start value of the next keyframe is always equal to the end value of the previous keyframe, the total length of the list is numFrames + 1. The specific storage rules of this data list vary according to the basic data type, as shown in the following table:

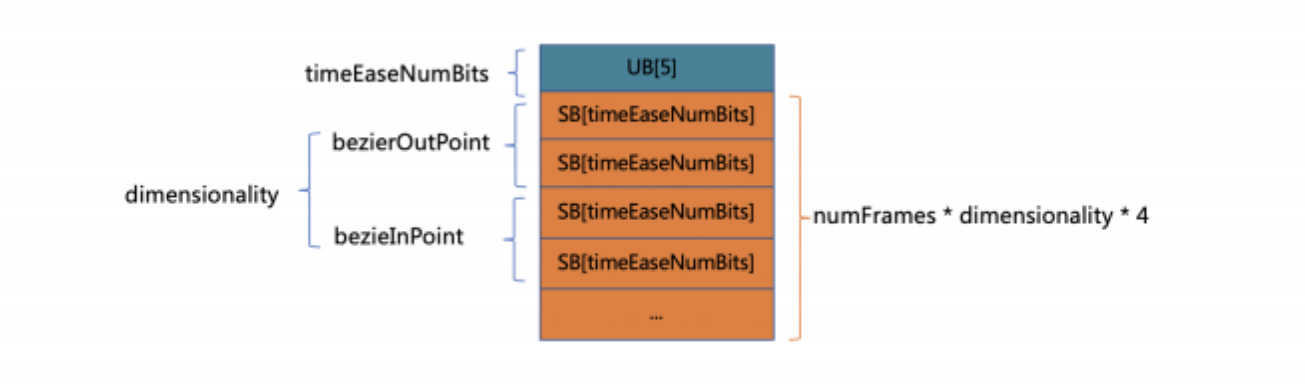
PropertyValue Storage Method

Type	Storage Method
Float[numFrames + 1]	Store numFrames + 1 Float data respectively in turn.
Bool[numFrames + 1]	Store numFrames + 1 bits in turn, each bit represents a Bool value.
Uint8[numFrames + 1]	Compressed and stored as 32-bit continuous unsigned integers, first store numBits of UB[5], and then store numFrames + 1 UB[numBits] data in turn; each data point represents the original Uint8 value.
Uint32[numFrames + 1]	Compressed and stored as 32-bit continuous unsigned integers, first store numBits of UB[5], and then store numFrames + 1 UB[numBits] data in turn; each data point represents the original Uint32 value.
Point[numFrames + 1]	Usually, numFrames + 1 Point data are stored sequentially. If the property type is **SpatialProperty**, divide the two Floats of each Point data by SPATIAL_PRECISION to convert it into a Uint32 list with a length of (numFrames+1)*2, and then compress and store it as a 32-bit continuous unsigned integer. The storage rules are the same as above.
Time[numFrames + 1]	Store numFrames + 1 EncodedUint 64 data in turn, each data representing the original Time data.

ID[numFrames + 1]	Store numFrames + 1 EncodedUint 32 data in turn , each data represents the original ID data .
Color[numFrames + 1]	Store numFrames + 1 Color data in turn.
Ratio[numFrames + 1]	Store numFrames + 1 Ratio data in turn.
String[numFrames + 1]	Store numFrames + 1 String data in turn.
Path numFrames + 1]	Store numFrames + 1 Path data in turn.
TextDocument[numFrames + 1]	Store numFrames + 1 TextDocument data in turn.
GradientColor[numFrames + 1]	Store numFrames + 1 GradientColor data in turn.

TimeEaseValueList

The storage structure of **TimeEaseValueList** is as follows:

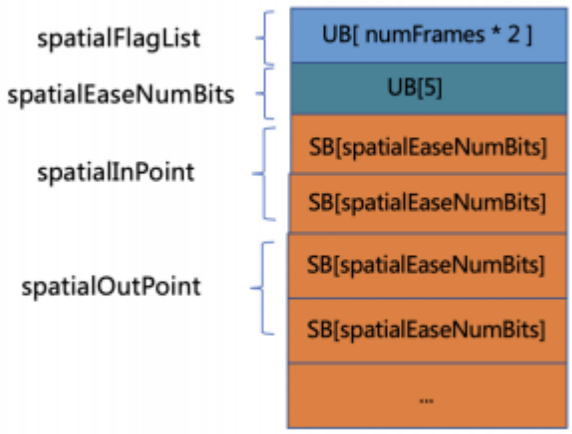


The reading of time easing parameters not only relies on the previous **timeEaseNumBits**, but also relies on a **dimensionality** parameter. Dimensionality represents the dimensions of the bezierIn and bezierOut arrays for each keyframe. It can be deduced from the number of components of the property type and the basic data type. For example, when the property type is **MultiDimensionProperty**, for attributes with a data type of Point, dimensionality is 2; if the property type is not **MultiDimensionProperty**, dimensionality is always 1. Each item in the timeEaseList list represents a Float component of the time easing parameter coordinate point. Two Floats form a Point. Usually, each keyframe of a one-dimensional time easing property has two Points, bezierIn and bezierOut, which means that the 4 Float components in timeEaseList need to be expressed sequentially. In the case of multi-dimensionality, all dimension time easing parameter arrays of this frame are read sequentially, and then the data of the next frame is read. In addition, if the current keyframe interpolation type is not **Bezier**, the process of reading time easing parameters for this frame will be skipped. The specific parsing code is as follows:

```
int dimensionality = config.attributeType == AttributeType::MultiDimensionProperty ?
config.dimensionality() : 1;
auto numBits = byteArray->readNumBits();
for (auto& keyframe:keyframes) {
if (keyframe->interpolationType != KeyframeInterpolationType::Bezier) {
continue;
}
float x, y;
for (int i = 0; i < dimensionality; i++) {
x = byteArray->readBits(numBits) * BEZIER_PRECISION;
y = byteArray->readBits(numBits) * BEZIER_PRECISION;
keyframe->bezierOut.emplace_back(x, y);
x = byteArray->readBits(numBits) * BEZIER_PRECISION;
y = byteArray->readBits(numBits) * BEZIER_PRECISION;
keyframe->bezierIn.emplace_back(x, y);
}
}
```

Spatial EaseValueList

The storage structure of **SpatialEaseValueList** is as follows:



Note: If the property type is not **SpatialProperty** or **AttributeFlag.hasSpatial** is false, this block does not need to be read. The reading of spatial easing parameters depends on the previously read **spatialFlagLis**t and **spatialEaseNumBits**. The spatialFlagList is twice as long as the number of keyframes because the spatial easing parameters of each keyframe contain two Point data. The values in the spatialFlagList list in turn indicate whether the spatialIn and spatialOut of each keyframe exist. If not present, use the default (0, 0) point. In addition, the read data is an integer, which needs to be multiplied by **SPATIAL_PRECISION** and converted to Float as the x and y data components of Point. The specific parsing code is as follows:

```
int index = 0;
for (auto& keyframe: keyframes) {
auto hasSpatialIn = spatialFlagList[index++];
auto hasSpatialOut = spatialFlagList[index++];
if (hasSpatialIn) {
                                        keyframe->spatialIn.x = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION;
                                        keyframe->spatialIn.y = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION;
}
if (hasSpatialOut) {
                                       keyframe->spatialOut.x = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION
                                       keyframe->spatialOut.y = byteArray->readBits(spatialEaseNumBits) * SPATIAL_PRECISION
}
}
```

Chapter 6: Label list

PAG uses 10 bits to store TagCode and can store up to 1024 kinds of Tags. Among them, 52 kinds of tags have been used, and the list is as follows:

TagCode Types

Name	**Value (Decimal)**	**Remarks**
End	0	Tag end identifier
FontTables	1	Font collection, containing multiple fonts
VectorCompositionBlock	2	Vector combination information
CompositionAttributes	3	The basic attribute information of the composition
ImageTables	4	Image collection information
LayerBlock	5	Layer information
LayerAttributes	6	The basic attribute information of layer
SolidColor	7	Border color
TextSource	8	Text information, including: text , font, size, color and other basic information
TextPathOption	9	Text drawing information , including: drawing path, front, back, left, and right spacing, etc.
TextMoreOption	1 0	Text additional information
ImageReference	1 1	Image reference , pointing to an image
CompositionReference	1 2	Composition reference , pointing to a composition
Transform2D	1 3	2D transform information
MaskBlock	1 4	Mask information

ShapeGroup	1 5	Shape information
Rectangle	1 6	Rectangle information
Ellipse	1 7	Ellipse information
PolyStar	1 8	Polygonal star
ShapePath	1 9	Shape path information
Fill	2 0	Fill rule information
Stroke	2 1	Stroke
GradientFill	2 2	Gradient fill
GradientStroke	2 3	Gradient stroke
MergePaths	2 4	Merge paths
TrimPaths	2 5	Trimming paths
Repeater	2 6	Repeater
RoundCorners	2 7	Round corners
Performance	2 8	File performance information, which is used to check whether PAG file performance meets the standard.
DropShadowStyle	2 9	Drop shadow
BitmapCompositionBlock	4 5	Bitmap sequence frame
BitmapSequence	4 6	Bitmap sequence
ImageBytes	4 7	Image byte stream
ImageBytes2	4 8	Image byte stream (with scaling)
ImageBytes3	4 9	Image byte stream (with transparent channel)
VideoCompositionBlock	5 0	Video sequence frame
VideoSequence	5 1	Video sequence

End

The End tag marks the end of the TAG. When the decoder reads this tag, it means that the content of this TAG has been read. If the Tag contains nesting, when encountering the End mark, you need to jump out of the current Tag to read and transfer to the outer Tag reading logic.

End Structure Table

Type	**Type**	**Remarks**
End	Uint16	Tag end identifier

FontTables

Font Tables is a collection of font information.

Font Tables Structure Table

Field	Type	Remarks
count	EncodedUint32	Number of fonts
fontData	FontData[]	Font array

VectorCompositionBlock

VectorCompositionBlock is a collection of vector graphics. It can contain simple vector graphics, and can also contain one or more VectorComposition.

Vectorcompositionblock Structure Table

Field	Type	Remarks
id	EncodedUint32	Unique identifier
TagBlock	TagBlock	TagBlock data block, refer to Chapter 3 TagBlock . Include two TAGs: CompositionAttributes (basic attribute information of the composition), LayerBlock (layer information)

CompositionAttributes

CompositionAttribute stores the basic attribute information of Composition. It can contain simple vector graphics, and can also contain one or more VectorComposition

Field	Type	Remarks
width	EncodedInt32	layer width
height	EncodedInt32	layer height
duration	EncodedUint64	duration
frameRate	Float	frame rate
backgroundColor	Color	background color

ImageTables

ImageTables is a collection of image information.

ImageTables Structure Table

Field	Type	Remarks
count	EncodedInt32	number of images
images	ImageBytes[count]	image array

LayerBlock

LayerBlock is a collection of layer information.

LayerBlock Structure Table

Field	Type	Remarks
type	Uint8	Layer type
id	EncodedUint32	unique identifier of layer
Tag Block	TagBlock	TagBlock data block, refer to Chapter 3 TagBlock Contains Tags: LayerAttributes, LayerAttributes2, MaskBlock, Transform2D, SolidColor, TextSource, TextPathOption, TextMoreOption, CompositionReference, ImageReference, etc.

LayerAttributes

LayerAttributes is the attribute information of the layer.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
isActive	Bool	BitFlag	true	If false , it will not be rendered.
autoOrientation	Bool	BitFlag	false	Adaptive screen ratio
parent	EncodedUint32	Value	0	Layer ID
stretch	Ratio	Value	(1,1)	Stretch ratio
startTime	Time	Value	0	Start time
blendMode	Enumeration (Uint8)	Value	BlendMode:: Normal	Layer blend mode
trackMatteType	Enumeration (Uint8)	Value	TrackMatteType::None	Track mask
timeRemap	Float	SimpleProperty	0.0f	
duration	Time	FixedValue	0	Time interval

SolidColor

SolidColor identifies the border width height and color attribute information.

SolidColor Structure Table

Field	Field Type	Remarks
solidColor	Color	Color value
width	EncodedInt32	Width
height	EncodedInt32	Hight

TextSource

TextSource indicates text information, including text, font, size, color and other basic information.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
sourceText	TextDocument	DiscreteProperty	TextDocument	

TextPathOption

TextPathOption indicates text drawing information, including drawing path, front, back, left, and right spacing, etc.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remark
path	EncodedUint32	Value	0	Mask ID
reversedPath	Bool	DiscretePro perty	false	
perpendicularToPath	Bool	DiscreteProperty	false	
forceAlignment	Bool	DiscreteProperty	false	
firstMargin	Float	SimpleProperty	0.0f	
lastMargin	Float	SimpleProperty	0.0f	

TextMoreOption

TextMoreOption indicates additional information about the text.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
anchorPointGrouping	Enumeration (Uint8)	Value	ParagraphJustify cation::LeftJustify	
groupingAlignment	Point	MultiDimensionProperty	(0 .0)	

ImageReference

ImageReference is the image reference tag, which stores the unique ID of an image and can index real image information by ID.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
id	EncodedUint32			

CompositionReference

CompositionReference is the layer combination index tag, which stores the unique ID of a layer composition and can index real layer composition by ID.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
id	EncodedUint32			
copositionStartTime	Time			start time

Transform2D

Transform2D indicates the 2D transformation information, including anchor point, scaling, rotation, x-axis offset, y-axis offset and other information.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
anchorPoint	Point	Value	(0.0)	anchor point
position	Point	Value	(0.0)	location information
xPosition	Float	Value	0.0	x-axis offset
yPosition	Float	Value	0.0	y-axis offset
scale	Point	Value	(0.0)	scaling
rotation	Float	Value	0.0	rotation
opacity	Uint8	Value	255	Transparency (0~255)

Mask

Mask tags.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
id	EncodedUint32	FixedValue	0	
inverted	Bool	BitFlag	false	
maskMode	Enumeration (Uint8)	Value	MaskMode::Add	
maskPath	Path	SimpleProperty		
maskOpacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)
maskExpansion	Float	SimpleProperty	0.0f	

ShapeGroup

ShapeGroup indicates the drop shadow tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
blendMode	Enumeration (Uint8)	Value	BlendMode::Normal	
anchorPoint	Point	SpatialProperty	(0.0)	anchor point
position	Point	SpatialProperty	(0.0)	location information
scale	Point	MultiDimensionProperty	(1, 1)	scaling
skew	Float	SimpleProperty	0.0	
skewAxis	Float	SimpleProperty	0.0	y-axis offset
rotation	Float	SimpleProperty	0.0	rotation
opacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)
TagBlock	TagBlock			TagBlock data block, refer to Chapter 3 TagBlock.

Rectangle

Rectangular tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
reversed	Bool	BitFlag	false	
size	Point	MultiDimensionProperty	(100, 100)	Width and height
position	Point	SpatialProperty	(0,0)	Location
roundness	Float	SimpleProperty	0.0f	

Ellipse

Ellipse tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
reversed	Bool	BitFlag	false	
size	Point	MultiDimensionProperty	(100,100)	Width and height
position	Point	SpatialProperty y	(0,0)	Location

PolyStar

Polygonal star tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks

reversed	Bool	BitFlag	false	
polyType	Enumeration (Uint 8)	Value	PolyStarType::Star	
points	Float	SimpleProperty	5.0f	
position	Point	SpatialProperty y	(0,0)	Location
rotation	Float	SimpleProperty	0.0f	
innerRadius	Float	SimpleProperty	50.0f	
outerRadius	Float	SimpleProperty	100.0f	
innerRoundness	Float	SimpleProperty	0.0f	
outerRoundness	Float	SimpleProperty	0.0f	

ShapePath

ShapePath tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
shapePath	Path	SimpleProperty		

Fill Tags

Fill tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
blendMode	Enumeration (Uint8)	Value	BlendMode::Normal	
composite	Enumeration (Uint8)	Value	CompositeOrder: : BelowPreviousInSameGroup	
fillRule	Enumeration (Uint8)	Value	FillRule::NonZeroWinding	
color	Color	SimpleProperty	Red	
opacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)

Stroke

Stroke tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
blendMode	Enumeration(Uint8)	Value	BlendMode::Normal	

composite	Enumeration (Uint8)	Value	CompositeOrder: : BelowPreviousInSameGroup	
lineCap	Enumeration (Uint8)	Value	LineCap::Butt	
lineJoin	Enumeration (Uint8)	Value	LineJoin::Miter	
miterLimit	Float	Simple Property	4.0f	
color	Color	Simple Property	White	
opacity	Uint8	Simple Property	255	Transparency (0 ~ 255)
strokeWidth	Float	SimpleProperty	2.0f	

GradientFill

GradientFill tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
blendMode	Enumeration (Uint8)	Value	BlendMode::Normal	
composite	Enumeration (Uint8)	Value	CompositeOrder: : BelowPreviousInSameGroup	
fillRule	Enumeration (Uint8)	Value	FillRule: : NonZeroWinding	
fillType	Enumeration (Uint8)	Value	GradientFillType: : Linear	
startPoint	Point	SpatialProperty	(0,0)	start point
endPoint	Point	SpatialProperty	(100,0)	end point
colors	Color[]	SimpleProperty		
opacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)

GradientStroke

GradientStroke tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
blendMode	Enumeration (Uint8)	Value	BlendMode::Normal	
composite	Enumeration (Uint8)	Value	CompositeOrder: : BelowPreviousInSameGroup	
fillType	Enumeration (Uint8)	Value	GradientFillType: : Linear	

startPoint	Point	SpatialProperty	(0,0)	start point
endPoint	Point	SpatialProperty	(100 ,0)	end point
color	Color	SimpleProperty	White	
opacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)
strokeWidth	Float	SimpleProperty	2.0f	
lineCap	Enumeration (Uint8)	Value	LineCap::Butt	
lineJoin	Enumeration (Uint8)	Value	LineJoin::Miter	
miterLimit	Float	SimpleProperty	4.0f	
dashLength	UB[3]			dashLength = UB[3] + 1
dashOffsetFlag . exist	UB[1]			Whether to contain the dashOffset tag
dashOffsetFlag.animatable	UB[1]			if dashOffsetFlag.exist= 1

MergePaths

MergePaths tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
mode	Enumeration (Uint 8)	Value	MergePathsMode::Add	

TrimPaths

TrimPaths tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
start	Float	SimpleProperty	0.0f	
end	Float	SimpleProperty	100.0f	
offset	Float	SimpleProperty	4.0f	
trimType	Enumeration (Uint8)	Value	TrimPathsType: : Simultaneously	

Repeater

Repeater tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks

composite	Enumeration (Uint8)	Value	CompositeOrder: : BelowPreviousInSameGroup	
copyes	Float	SimpleProperty	3.0f	
offset	Float	SimpleProperty	0.0f	
anchorPoint	Poi nt	SpatialProperty	(0,0)	
position	Point	SpatialProperty	(100,0)	
scale	Point	MultiDimensionProperty	(1.1)	scaling
rotation	Float	SimpleProperty	0.0f	
startOpacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)
end Opacity	Uint8	SimpleProperty	255	Transparency (0 ~ 255)

RoundCorners

Round Corners tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
radius	Float	SimpleProperty	10.0f	

Performance

Performance tag mainly stores PAG performance index data.

Performance Structure Table

Field	Field Type	Remarks
renderingTime	EncodedInt 64	Time cost of rendering
imageDecodingTime	EncodedInt 64	Time cost of decompression
presenttingTime	EncodedInt 64	
graphicsMemory	EncodedInt 64	Rendering memory

DropShadowStyle

DropShadowStyle tag.

AttributeBlock Structure Table

Field	Field Type	Attribute Type	Defaults	Remarks
blendMode	Enumeration (Uint8)	DiscreteProperty	BlendMode::Normal	
color	Color	SimpleProperty	Black	
opacity	Uint8	SimpleProperty	191	Transparency (0 ~ 255)

angle	Float	SimpleProperty	120.0f	
distance	Float	SimpleProperty	5.0f	
size	Float	DiscreteProperty	5.0f	

BitmapCompositionBlock

BitmapCompositionBlock is the tag of the bitmap sequence frame.

AttributeBlock Structure Table

Field	Field Type	Remarks
id	EncodedUint32	unique identifier
TagBlock	TagBlock	TagBlock data block, refer to Chapter 3 TagBlock . Contain two TAGs: CompositionAttributes (basic attribute information of composition) and BitmapSequence (bitmap information)

BitmapSequence

BitmapSequence tag.

BitmapSequence Structure Table

Field	Field Type	Remarks
width	EncodedUint32	
height	EncodedUint32	
frameRate	Float	
frameCount	EncodedUint 32	Number of bitmap frames
isKeyFrameFlag[frameCount]	UB[frameCount]	frameCount flags whether they are keyframes
bitmapRect[frameCount]	BitmapRect[frameCount]	frameCount data of BitmapRect

ImageBytes

ImageBytes is a kind of image tag that stores compressed image-related attribute information.

ImageBytes Structure Table

Field	Field Type	Remarks
id	EncodedUint32	
fileBytes	ByteData	image byte stream

ImageBytes2

ImageBytes2 is version 2 of the image tag, which not only stores the information of ImageBytes but also allows the recording of the scaling parameters of the image. Usually, the image is stored according to the actual maximum size used, not the original size.

ImageBytes 2 Structure Table

Field	Field Type	Remarks
id	EncodedUint32	
fileBytes	ByteData	image byte stream
scaleFactor	Float	scaling ratio (0 ~ 1.0)

ImageBytes3

ImageBytes3 is version 3 of the image tag. In addition to containing the information of ImageBytes2, it also allows recording the image after removing the transparent border.

Field	Field Type	Remarks
id	EncodedUint32	
fileBytes	ByteData	image byte stream
scaleFactor	Float	scaling ratio (0 ~ 1.0)
width _	EncodedInt32	original picture width
width	EncodedInt32	original picture width
anchorX	EncodedInt32	The x-axis coordinate of the starting point of the opaque area in the original image
anchorY	EncodedInt32	The y-axis coordinate of the starting point of the transparent area in the original image

VideoCompositionBlock

VideoCompositionBlock stores one or more video sequence frames of different sizes.

VideoCompositionBlock Structure Table

Field	Field Type	Remarks
id	EncodedUint32	unique identifier
hasAlpha	Bool	Whether there is an Alpha channel
CompositionAttributes	CompositionAttributesTag	
TagBlock	TagBlock	TagBlock data block, refer to Chapter 3 TagBlock . Contains two TAGs: CompositionAttributes (basic attribute information of composition) and VideoSequence (bitmap information)

VideoSequence

VideoSequence stores 1 version of the video sequence frame structure.

VideoSequence Structure Table

Field	Field Type	Remarks
width	EncodedUint32	
height	EncodedUint32	
frameRate	Float	
alphaStartX	EncodedInt32	This value will only exist if the hasAlpha read from VideoCompositionBlock is 1.
alphaStartY	EncodedInt32	This value will only exist if the hasAlpha read from VideoCompositionBlock is 1.
spsData	ByteData	The byte stream that has been read does not contain (0, 0, 0, 1) four-byte Start Code
ppsData	ByteData	The byte stream that has been read does not contain (0, 0, 0, 1) four-byte Start Code
frameCount	EncodedUint32	Number of bitmap frames
isKeyFrameFlag [frameCount]	UB[frameCount]	frameCount flags whether they are key frames
videoFrames	VideoFrame[frameCount]	Number of video frames