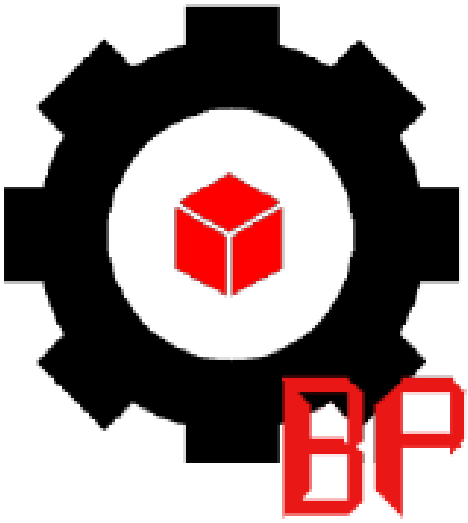
BPX Format

https://github.com/BlockProject3D/

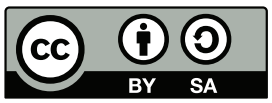
Yuri Edward

04-28-2020



Abstract

This document describes the BPX format, providing an optimized, flexible and DRM [10] free storage designed for various multimedia content types commonly used in 3D applications.



1 Introduction

When developing a 3D game engine often comes the problem of dealing with game modifications (mods). For example one might consider how easy would it be for the end player to add custom assets, change existing assets; would the end player need to agree against certain complex licences or if it is even legally allowed to modify the content of certain asset files.
Here the term asset refers to various type of file loaded by the game for a specific game world or for the entire game. These resources can be multimedia files, scripts, configurations and other formats the game engine would use.
This format provides an alternative to existing asset formats geared towards the development of highly moddable (modify the game; add new assets, remove existing assets, change existing assets) and cross platform 3D games.
The benefits are:

- Open source with a permissive licence.
  Redistribute source code, allow improvements of the format by others.
- DRM [10] free.
  Avoid licensing problems and restrictions. DRM stands for Digital Rights Management and is used to control how the user is allowed to interact with certain multimedia content. In moddable game DRM usually restricts the user from modifying existing assets.
- Flexible.
  Designed for providing support for moddability. The nature of container allows support for all kind of multimedia resources used in a typical real-time 3D application.
- Optimized for 3D APIs such as OpenGL [4], DirectX [7], Vulkan [5]
  Certain types of BPX uses some assumptions on the input data in order to reduce pre-process time when loading assets in a typical real time rendering engine.
- Cross platform.
  Can store data for compatibility over multiple platforms within a single file due to its nature of containers.

2 History

The BPX format also refered to BlockProject Extended is a remake of an old format designed for the first version of the BlockProject 3D engine as an attempt to allow moddability, that means change the behaviour of a game while it's running and in this case change also any multimedia asset including scenes, shaders, models and even textures...
This has proved to be a difficult task due to licensing issues, restrictions imposed by existing frameworks and restrictions imposed by certain formats.
As a result a custom asset format has been developed for initially a very limited set of type of assets in order to solve the issue.
This format is an evolution geared towards re-usability and performance.

3 Comparison against mainstream formats

Below is a non-exhaustive table showing the differences between the idea of BPX and existing mainstream formats:

Name	Open	DRM	Flexible	Optimized	Compat.
FBX	No	No	No	Yes	Partial
Unreal	No	Yes	Yes	Yes	Partial
Unity	No	Yes	Partial	Yes	Partial
OBJ	Yes	No	No	No	Yes

- FBX: It is possible to save files as Text based FBX however no open documentations exists for the format. The binary FBX requires the use of a closed source library FBX supports Windows without issues however getting the SDK to work on different platforms might be an issue.
  FBX is designed to support 3D models.

- The source code of Unreal can be accessed if you are agree to certain licences which prohibits the redistribution of the editor source code in the game preventing the use of Unreal's format as a base to support modding. Unreal does also not provide easy installations under Linux.
  Unreal is also using a unified format for all assets of the game. It has support for modding however the modder is required to agree to licences and to download the full SDK.

- Unity although it has better support for Linux it might still be a bit tricky in certain combinations of distribution/system. The way unity has
  Unity is also using a unified format for all assets of the game but the game engine does not allow load of arbitrary custom assets. The asset format used by unity is also closed and does not permit modification by the player of the game like Unreal.

- OBJ is a text based 3D model format. The format has no support for skeletal based models/animations and has poor support for materials.

4 General

The format is based on the idea of a container with a reusable main header that can be extended for the different applications.
The following assumptions are used:

- A byte is assumed to be 8 bits.
- The sizes are expressed in bits in the entire document.
- The byte order in a BPX file should be **little endian** to match most of current hardware/software architectures, and then avoid a pre-processing step when loading the file.
- A transformation is stored in terms of float to match most current rendering APIs.
- The coordinate system used for this format is assumed to be left handed with the Z axis to be pointing upward.

Description of the coordinate system:

$$Right = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} Forward = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} Up = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad (1)$$

The TypeExt section available in certain BPX types describes what fields to expect in the 128 bits of extension located in the BPX Main Header.
If the TypeExt field isn't used by a specific BPX type then this field should be set to 0.

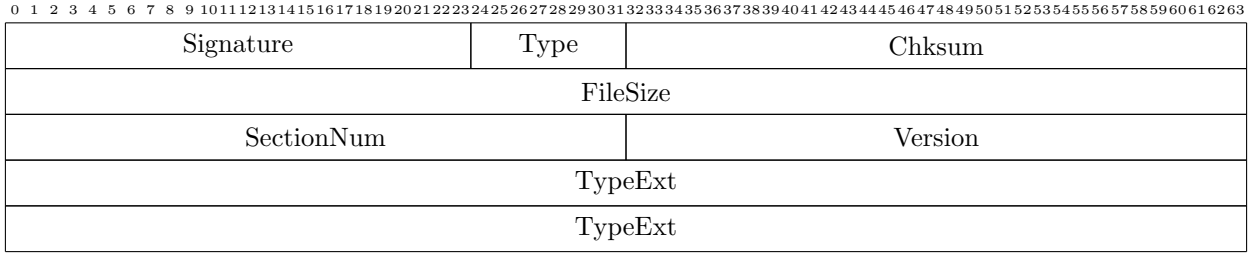
5 Padding rules

All data structures exposed in this format specification are padded to comply with major C/C++ compilers: Visual C++, GCC and Clang.

6 BPX Main Header

The BPX Main Header describes general information about the container and the contained data.
This section is **not compressable**.
Below is a table describing the different fields to be expected in the header:

Name	Type	Size	Notes
Signature	String	24	File signature; always "BPX"
Type	Unsigned	8	Type of BPX
Chksum	Unsigned	32	Checksum
FileSize	Unsigned	64	Size of file after compression
SectionNum	Unsigned	32	Number of sections
Version	Unsigned	32	Version of format
TypeExt	Unspecified	128	Extension space for various BPX types



6.1 Signature

Three characters to describe the file when open in a text/hex editor

6.2 Type

The type of BPX. This is used to describe what type of sections to expect in the file.
Currently only the following are supported:

- 'T' for Texture
- 'M' for Model
- 'S' for Shader
- 'C' for Scene
- 'P' for Package

Here the characters between single quotes are to interpreted as their byte representation in ASCII encoding.

6.3 Version

Version of the file, the currently only available version of the format is 0.

6.4 SectionNum

Number of sections in the file.

6.5 Chksum

Checksum calculated with the help of a CRC32 algorithm. This should integrate both the header (except for the Chksum field) and the file content after compression.

6.6 FileSize

The file size field corresponds to the total size of the file minus the size of the header in bytes after compression; this is used as an additional security over the integrity of the file. It can also be used to check the remaining number of bytes in a network based streaming application.

6.7 TypeExt

The TypeExt field provides 64 bits of extension for different kind of BPX avoiding having to store or load additional sections in order to get usefull general information about a specific file reducing load time and memory complexity.
All unused space in this field should be 0.

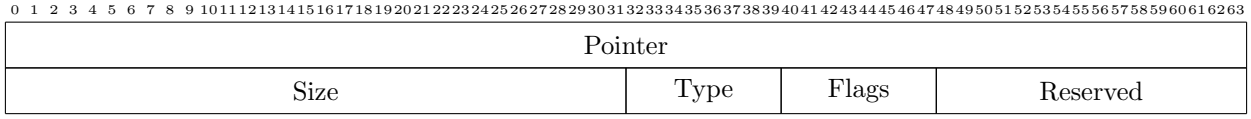
7 BPX Section Header Table

The BPX Section Header Table is an array of data structures describing important information for each section present in the file. It is located after the BPX Main Header.
The size of this array is determined by the SectionNum field in the BPX Main Header. The array is expected to be contiguous.
Certain types of BPX expects to be able to index sections in this table using positions. For that reason, the array is expected to be 1-indexed to allow for a value of 0 to represent a null or non-existent section.
This section is **not compressable**.

7.1 BPX Section Header

Below is a table describing the feilds to expect in a BPX Section Header:

Name	Type	Size	Notes
Pointer	Unsigned	64	Offset in bytes from the beginning of the file
Size	Unsigned	32	Size in bytes of section
Type	Unsigned	8	Type of Section
Flags	Unsigned	8	Flags for the current section
Reserved	Unspecified	16	Blank, always 0



7.1.1 Size

Size in bytes of the content for a given section excluding Section Header.

7.1.2 Pointer

Position in bytes in the file where the content for a given section should be found.

7.1.3 Type

An integer to represent the type of section.

7.1.4 Flags

Bit mask based flags (or flags together). Currently the only supported flags are:

Name	Value	Notes
Compressed	0x2	Indicates the section is compressed using the zlib [6] algorithm

8 BPX Type: Texture ('T')

8.1 Overview

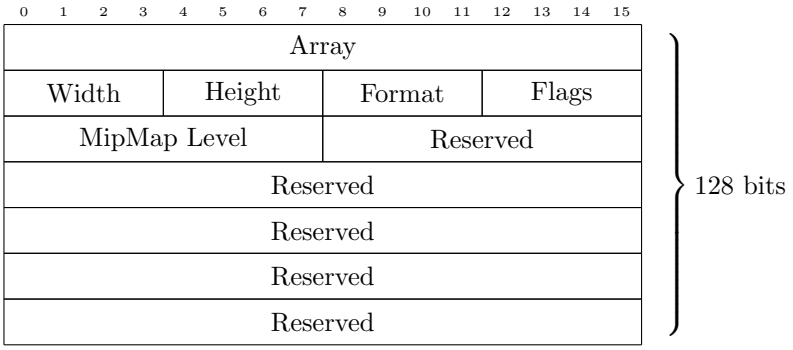
The Texture BPX is using 'T' as the type byte of BPX Main Header. This type provides optimized and efficient texture storage for 3D rendering APIs.
Below is a table describing the different sections to be expected in a BPXT:

Name	Type	Required	Single Time
PixelArray	1	Yes	Yes

8.2 TypeExt

Contains information about the texture encoding.

Name	Type	Size	Notes
Array	Unsigned	16	Size of texture array
Width	Unsigned	4	Texture width
Height	Unsigned	4	Texture height
Format	Unsigned	4	Texture format
Flags	Unsigned	4	Flags
MipMap Level	Unsigned	8	Number of mip maps
Reserved	Unspecified	72	Blank, always 0



8.2.1 Width

The expected width of all pixel arrays in power of two form (2^{k+1}px where k is the stored number).

8.2.2 Height

The expected height of all pixel arrays in power of two form (2^{k+1}px where k is the stored number).

8.3 Analysis on texture size storage

Certain redering APIs requires that the textures are aligned to a specific implementation defined number of pixels per row. This number also called stride is usualy 8 or a power of two greater than 8.
By assuming any BPX encoded texture is encoded with power of twos instead of their actual resolution in pixels, we can eliminate the need, in the cases where the implementation uses power of two strides, to run a

padding alignment before presenting the texture to this implementation. Also this allows to reduce the field size for storing texture size: instead of using 32 bits or 64 bits we can store a texture size in 8 bits and keep relatively large texture sizes, **optimizing both storage space and application load speed**. Indeed the maximum texture size in each direction allowed by BPX would be $2^{15+1} = 2^{16} = 65536$. Most rendering API implementation do not support such large texture sizes.
From there one might say that we should then use less than 8 bits. However using less than 8 bits for storing the entire size vector would conflict with hardware indexing as most hardware only indexes 8 bits bytes.

8.3.1 Format

Available formats:

Name	Value	Notes
RGB	0x1	Standard 8 bits 3 channel RGB format
RGBA	0x2	8 bits 4 channel RGB with transparency level
GREY_SCALE	0x3	Single 8 bits channel representing grey scale level
FLOAT	0x4	Single channel 32 bits float texture

8.3.2 Flags

Bit mask based flags (or flags together). Currently the only supported flags are:

Name	Value	Notes
Array	0x1	Indicates the texture should be interpreted as a texture array
Compressed	0x2	Indicates the texture should be GPU compressed on load
CubeMap	0x4	Indicates the texture should be interpreted as a CubeMap

8.3.3 MipMap Level

Number of mip maps [2] to auto generate when loading this texture.

8.3.4 Array

Number of textures for creating a texture array. If this value is 0 consider this is not a texture array.
Ignore this value if neither of Array or CubeMap flags are set.

8.4 PixelArray

Texture data array encoded as described by texture format.
If the *Array* value in the header is greater than 0 then a series of *Array* count texture data arrays are saved one after the other encoded with respect to the texture format described by the header.
In case the CubeMap flag is set the *Array* field in the header should be 6 and this section should contain 6 texture data arrays.

9 BPX Type: Model ('M')

9.1 Overview

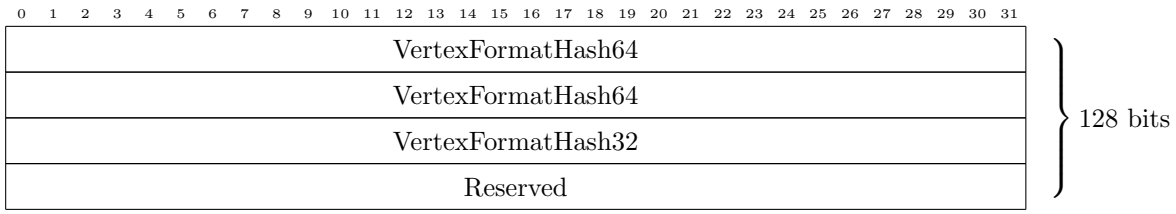
The Model BPX is using 'M' as the type byte of BPX Main Header. This type provides optimized and efficient mesh storage for 3D rendering APIs.
Below is a table describing the different sections to be expected in a BPXM:

Name	Type	Required	Single Time
VertexArray	2	Yes	No
IndexArray	3	No	No
BoneArray	4	No	Yes
FrameArray	5	No	Yes
AnimationArray	6	No	Yes
Strings	255	No	Yes

9.2 TypeExt

Contains general information about the 3D model file.

Name	Type	Size	Notes
VertexFormatHash64	Unsigned	64	Hash of vertex format
VertexFormatHash32	Unsigned	32	Hash of vertex format
Reserved	Unspecified	32	Blank, always 0



9.2.1 VertexFormatHash64

64 bits hash of vertex format asset virtual path corresponding to the vertex structure that this shader expects as input.

9.2.2 VertexFormatHash32

32 bits hash of vertex format asset virtual path corresponding to the vertex structure that this shader expects as input.

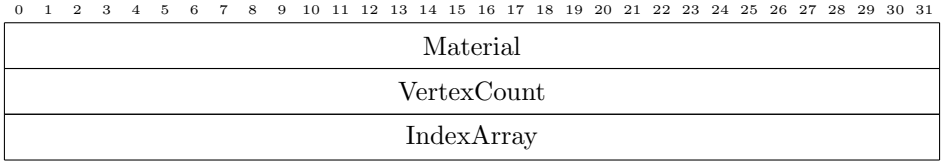
9.2.3 VertexFormat

The actual definition of the vertex format is specified as a separate asset, left to the implementation for flexibility.

9.3 VertexArray

This section contains a header followed by an array of vertex data structure. The size of each vertex structure should be equal to the field VertexSize in the VertexFormat section.
The header data structure is described in the following table:

Name	Type	Size	Notes
MaterialHash64	Unsigned	64	Hash of material
MaterialHash32	Unsigned	32	Hash of material
VertexCount	Unsigned	32	Total number of vertices
IndexArray	Unsigned	32	Index array section index



9.3.1 MaterialHash64

64 bits hash of material asset virtual path to apply by default to this vertex array.

9.3.2 MaterialHash32

32 bits hash of material asset virtual path to apply by default to this vertex array.

9.3.3 VertexCount

The amount of vertex structures to read after this header.
We remind that all modern hardware support triangles as primitives, so the vertex count should be a multiple of 3

9.3.4 IndexArray

This field is an index in the section header table to an IndexArray that should be used alongside this VertexArray.
A value of 0 means that no IndexArray is associated to that VertexArray.

9.3.5 Total array size

The total size in bytes of the vertex array to use as pre-allocation buffer size when loading this model can be calculated as follows:

$$SA = S \times V \tag{2}$$

where SA is the total size of the vertex array, S the size of a single vertex structure as given in the VertexFormat asset and V the number of vertices in the array given by the VertexCount field.
To avoid storing an additional 64 bits field in the header of the VertexArray section this size SA is left to be calculated by the implementing application.

9.4 IndexArray

The index array section allows to ommit certain vertices and figure out these vertices at runtime. Instead of writing all vertices for a given model requiring more space to store them, we only store unique vertices, and duplicates are just resolved by matching a vertex index from this section to an actual vertex structure, **this allows to save storage space and RAM**.
This section contains a header describing the amount of indices to be found in the array, and an array of 32 bits unsigned integer each representing the index of a vertex in a given vertex array.
Below is a table describing the data structure to expect as header for an IndexArray section:

Name	Type	Size	Notes
IndexCount	Unsigned	32	Total number of indices

9.4.1 IndexCount

The number of indices expected to be read from the array below this header.

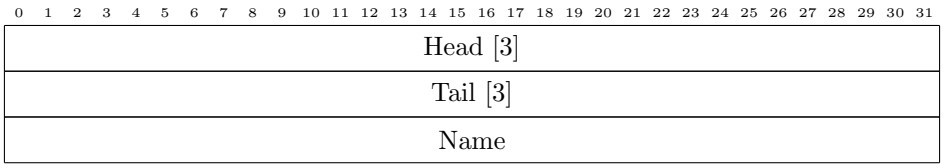
9.5 BoneArray

This section contains a header followed by an array of data structures representing each bone in a skeletal mesh.
Below is a description of the header to find before bone data in this section:

Name	Type	Size	Notes
BoneCount	Unsigned	16	Total number of bones

Each bone in the array following the header is represented by the following data structure:

Name	Type	Size	Notes
Head	3D float vector	96	Initial position of bone head
Tail	3D float vector	96	Initial position of bone tail
Name	Unsigned	32	Pointer in the Strings section



9.5.1 BoneCount

Number of bones to expect in the bone array following the header. This number should belong to the interval $[1, 1024]$.

9.6 Analysis on theoretical bone limit

We can give an estimation that human skeleton can be up to 400 bones. Of course most models do not include as many bones to reduce GPU/CPU load.

- OpenGL, Vulkan and Metal maximum UBO/function constants storage size available to a shader is usualy 64Kb or 65536 bytes

- On the official MSDN page [8] for DirectX the maximum constant buffer size available to a shader is $4096 constants \times 4 \times 32 bits = 4096 constants \times 16 bytes = 65536 bytes$

UBO, function constants and constant buffer refers to means of variable synchronisation between CPU accessible memory and GPU accessible memory. Vulkan is a special case as it actually supports faster memory called Push Constants, however these constants usually maxes at 256-512 bytes which is impracticable for storing bone transforms. Push constants are typically used to exchange important transformation matrices such as model, view or projection. The size of a transformation matrix in 3D space using homogeneous coordinate system [3] is $32 bits(float) \times 4 \times 4(4 \times 4 matrix) = 16 \times 32 bits = 512 bits = 64 bytes$.
So assuming the lowest size for storing bones in VRAM for GPU processing is 65536 we can theoretically store up to 1024 matrices assuming each bone matrix represents 3D space transformations in homogeneous coordinate system ($\frac{65536}{64} = 1024$).
Which means in order to **keep compatibility** with as many platforms/rendering apis as possible also taking into account byte indexing, we should store the bone count as a 16 bits integer.

9.6.1 Head

Initial head position for that specific bone.

9.6.2 Tail

Initial tail position for that specific bone.

9.6.3 Name

Bone name as a pointer to a null terminated string in the strings section.

9.7 FrameArray

This section contains a header followed by an array of data structures each representing the new transform for a combination of a bone and frame.
The array should be organized as the following diagram describes:

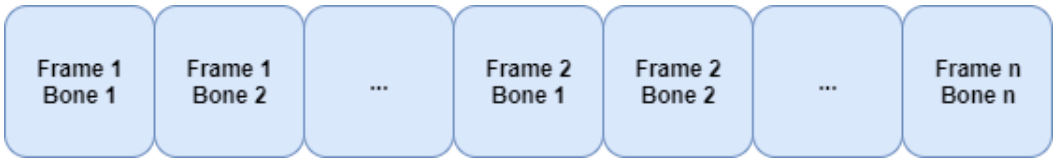


Figure 1: Diagram for organizing frame array

A single animation in a 3D model is usualy a set of actions. For example consider the following set of actions on a 3D human based model:

- move left leg
- move right leg
- move left arm
- move right arm

This set of actions could be grouped together in one animation named walking or running.
We call total animation time, the total time needed to express all animations for a single 3D model.

9.7.1 Design decision

The majority of mainstream formats represents animated models using key frames, that is storing only determinant changes in the model and interpolating intermediate frames at runtime (either when loading the file or while rendering). In BPX type M each frame is stored in the file to allow for any interpolation method, that is the interpolation if there's any is defined by the exporter of that model. The engine runtime does not have to provide any interpolation function for BPX type M.
This change grants the **flexibility** of using any interpolation functions even user defined.

9.7.2 Analysis on theoretical required storage size

We assume the total size required to store the full FrameArray is given by the following formula:

$$F \times B \times B_s \tag{3}$$

where F is the total amount of frames, B the number of bone updates per frame and B_s is the size of a single bone update structure.
We previously indicated that a bone update is determined by a transformation matrix in homogeneous coordinate space [3], that means it is exactly $512bits$ or $64bytes$.
Of course in a real application, it is very unlikely to have every frame in the timeline requiring update on every bone.

Worst case scenario
We assume the total animation time won't exceed 4 minutes.
We showed earlier that the theoretical maximum bone limit is 1024.
For the frame rate we choose 60, that means each second 60 frames in this FrameArray have to be rendered.
The total amount of frames to store is $60 \times 4 \times 60 = 14400frames$.
The total size required to save a model assuming each frame requires dependency over exactly all bones is $14400 \times 1024 \times 512 = 7549747200bits = 943718400bytes$. That is approximately $944Mb$.

Average case scenario
Of course having 1024 bones in one model is unlikely to happen. This is why we want to have an average estimation on the size required to store the FrameArray section.
On average, the amount of bones per animated model won't exceed 50 bones.
On average, each frame will update at most 10 bones.
On average, the frame rate of a 3D model animation will be $30FPS$.
On average, the total animation time won't exceed 2 minutes.
The total amount of frames to store is $30 \times 2 \times 60 = 3600frames$.
The total size required to save the FrameArray of a model assuming each frame requires dependency over exactly all bones is $3600 \times 10 \times 512 = 18432000bits = 2304000bytes$. That is approximately $2Mb$.

In conclusion the FrameArray section **should be compressed** to reduce the storage space consumed by a single model asset on disk. When loading the model in RAM one might consider adding some compression or supporting a lower limit than 1024 bones.

9.7.3 Bone transform

Each bone transformation in the FrameArray section is represented by the following structure:

Name	Type	Size	Notes
Rotation	4D float vector	128	Target bone rotation as a quaternion
Position	3D float vector	96	Target bone position
Scale	3D float vector	96	Target bone scale
BoneId	Unsigned	32	Index of affected bone

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Rotation [4]
Position [3]
Scale [3]
BoneId

9.8 Strings

The strings section contains a list of null-terminated strings to be referenced by start offset from other sections.

10 BPX Type: Shader Package ('S')

10.1 Overview

The Shader Package BPX is using 'S' as the type byte of BPX Main Header. This type provides optimized and cross API/platform storage for rendering code intended to be executed on the GPU [9].
Below is a table describing the different sections to be expected in a BPXS:

Name	Type	Required	Single Time
Shader	1	Yes	No
Assembly	2	Yes	No
Bindings	3	Yes	No
MaterialConstants	4	Yes	Yes
Strings	255	Yes	Yes

At least one assembly section is required to be saved in the file.

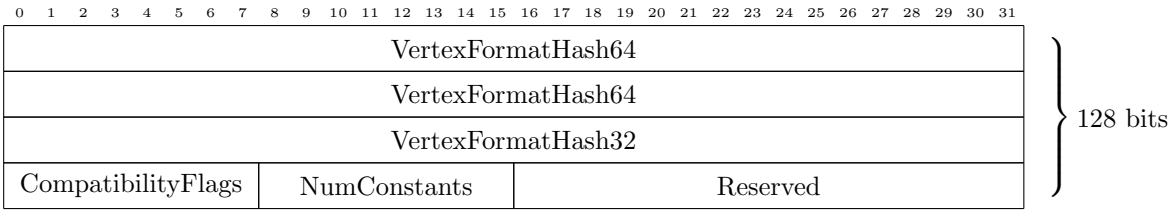
10.1.1 Design decisions

The shader type has been designed to store different versions of a single shader but conpiled against different rendering APIs.
For this reason, each rendering API should load the shaders composing the program by referring to it's Assembly section.
The assembly section also contains required information when loading a shader program.
In order to account for cases where it is not possible to provide version for each rendering API of a given shader, the shader program BPX contains a bit mask field to indicate compatibility or incompatibility against a given rendering API.
For example a shader compiled under Linux won't have DirectX shader byte code, it can store the source code HLSL (High Level Shading Language is Microsoft's shading language for DirectX) however not all DirectX enabled systems can compile shaders on the fly.
In conclusion, for **better compatibility** it is recommended to build shaders under Windows, for the **best compatibility** a Mac will be needed in order to build shaders for MSL (Metal Shading Language is Apple's shading language for Metal API).

10.2 TypeExt

Contains information about the shader package.

Name	Type	Size	Notes
VertexFormatHash64	Unsigned	64	Hash of vertex format
VertexFormatHash32	Unsigned	32	Hash of vertex format
CompatibilityFlags	Unsigned	8	Flags
NumConstants	Unsigned	8	Number of constants
Reserved	Unspecified	16	Blank, always 0



10.2.1 VertexFormatHash64

64 bits hash of vertex format asset virtual path corresponding to the vertex structure that this shader expects as input.

10.2.2 VertexFormatHash32

32 bits hash of vertex format asset virtual path corresponding to the vertex structure that this shader expects as input.

10.2.3 CompatibilityFlags

Bit mask based flags (or flags together). These flags are used to check if a given shader program can be used with the current configuration of hardware/driver/rendering implementation. By default packages compiled from BPSL will be compatible with OpenGL 3.0 and greater; DirectX 11 and 12 will be supported if the compiling machine has Windows and the DirectX SDK (the DirectX SDK is not available on Linux/Mac/etc). Currently the only supported flags are:

Name	Value	Notes
DirectX	0x1	Indicates compatibility with DirectX [7]
OpenGL	0x2	Indicates compatibility with OpenGL [4]
Vulkan	0x4	Indicates compatibility with Vulkan [5]
Metal	0x8	Indicates compatibility with Metal [1]

10.2.4 NumConstants

Number of constant description structures in the MaterialConstants section.

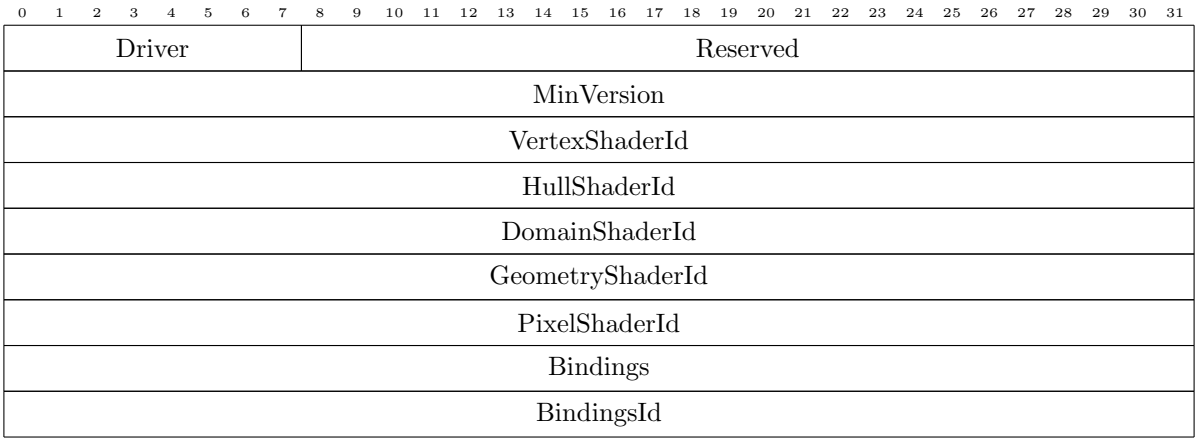
10.3 Shader

Contains shader data for a single rendering API...
A shader package can contain multiple shaders.

10.4 Assembly

Contains the shader list, order and other linking information for use by a rendering API. Below is a table describing the data structure to expect in this section:

Name	Type	Size	Notes
Driver	Unsigned	8	Target driver of this assenbly
Reserved	Unspecified	24	Blank, always 0
MinVersion	Unsigned	32	Minimum supported version
VertexShaderId	Unsigned	32	Vertex shader section index
DomainShaderId	Unsigned	32	Domain shader section index
HullShaderId	Unsigned	32	Hull shader section index
GeometryShaderId	Unsigned	32	Geometry shader section index
PixelShaderId	Unsigned	32	Pixel shader section index
Bindings	Unsigned	32	Number of bindings
BindingsId	Unsigned	32	Bindings section id



All indexes are given as positions in the Section Header Table.

10.4.1 Driver

The target rendering API for this assembly. List of possible values:

Name	Value
OpenGL	0
DirectX	1
Vulkan	2
Metal	3

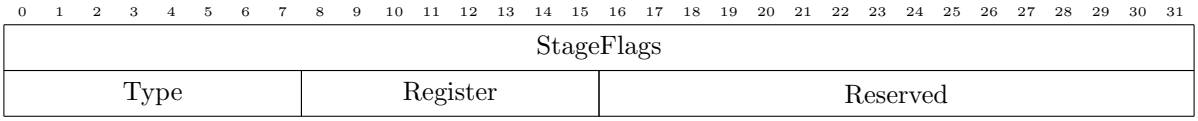
10.4.2 MinVersion

The minimum supported version for this shader. This value is dependent over the target rendering API supported by the assembly. Refer to the rendering API implementation for information on how the minimum version is encoded.

10.5 Bindings

The bindings section stores information about what bindings are used in this shader assembly for a particular rendering API.
This section stores an array of binding structures. A binding structure is defined by:

Name	Type	Size	Notes
StageFlags	Signed	32	Stage flags
Type	Unsigned	8	Type of binding
Register	Unsigned	8	Register number
Reserved	Unspecified	16	Blank, always 0



10.5.1 StageFlags

Describes to what stage this binding is acceptable. These flags are bit mask that can be or'ed together in order to map to multiple stages. Below is a table to list the different available flags:

Name	Value	Notes
LOCK_VERTEX_STAGE	0x1	Indicates binding locks to vertex shader
LOCK_HULL_STAGE	0x2	Indicates binding locks to hull shader
LOCK_DOMAIN_STAGE	0x4	Indicates binding locks to domain shader
LOCK_GEOMETRY_STAGE	0x8	Indicates binding locks to geometry shader
LOCK_PIXEL_STAGE	0x10	Indicates binding locks to pixel shader

10.5.2 Type

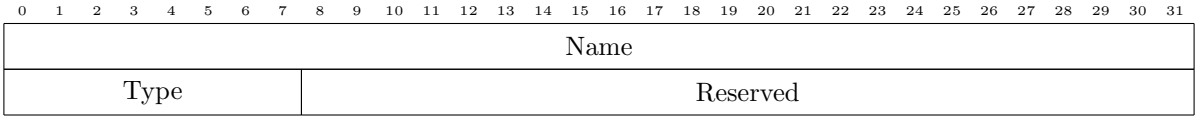
The type of binding. Currently there are only 4 types of binding:

Name	Value
TEXTURE	0
SAMPLER	1
CONSTANT_BUFFER	2
FIXED_CONSTANT_BUFFER	3

10.6 MaterialConstants

The Material structure expected by this shader. Like for the binding section, this section is also a contiguous array of constant description structures. A constant description structure is defined as follows:

Name	Type	Size	Notes
Name	Unsigned	32	Name of constant
Type	Unsigned	8	Type of constant
Reserved	Unspecified	24	Blank, always 0



10.6.1 Name

The name of the constant as an offset pointer in the string section.

10.7 Type

The type of constant. Below is an exhaustive table of all accepted types:

Name	Value
FLOAT	0
VECTOR_FLOAT_2	1
VECTOR_FLOAT_3	2
VECTOR_FLOAT_4	3
INT	4
VECTOR_INT_2	5
VECTOR_INT_3	6
VECTOR_INT_4	7
UINT	8
VECTOR_UINT_2	9
VECTOR_UINT_3	10
VECTOR_UINT_4	11
BOOL	12
VECTOR_BOOL_2	13
VECTOR_BOOL_3	14
VECTOR_BOOL_4	15
COLOR_RGB	16
COLOR_RGBA	17
DOUBLE	18
VECTOR_DOUBLE_2	19
VECTOR_DOUBLE_3	20
VECTOR_DOUBLE_4	21

10.8 Strings

The strings section contains a list of null-terminated strings to be referenced by start offset from other sections.

11 BPX Type: Package ('P')

11.1 Overview

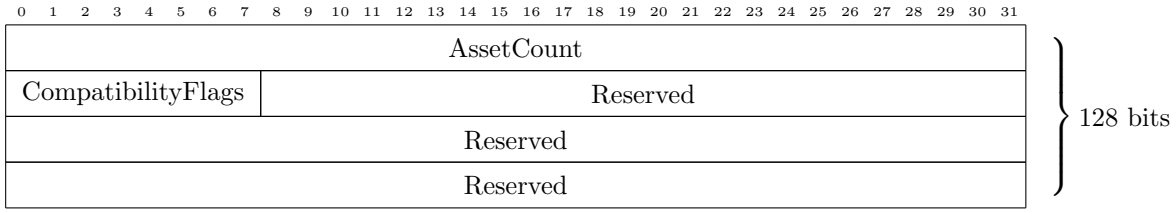
The Package BPX is using 'P' as the type byte of BPX Main Header. This type provides asset packages using the BPX format.
Below is a table describing the different sections to be expected in a BPXP:

Name	Type	Required	Single Time
AssetRegistry	1	Yes	Yes
File	2	Yes	No
Strings	255	Yes	Yes

11.2 TypeExt

Contains information about the package content.

Name	Type	Size	Notes
AssetCount	Unsigned	32	Number of assets
CompatibilityFlags	Unsigned	8	Information about supported env.
Reserved	Unspecified	88	Blank, always 0



11.2.1 AssetCount

Number of assets to expect in the AssetRegistry section.

11.2.2 CompatibilityFlags

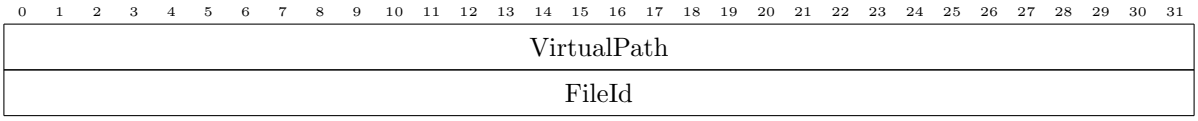
Bit mask based flags (or flags together). These flags are used to check if a given asset package can be used with the current configuration of hardware/driver/rendering implementation.
Currently the only supported flags are:

Name	Value	Notes
DirectX	0x1	Indicates compatibility with DirectX [7]
OpenGL	0x2	Indicates compatibility with OpenGL [4]
Vulkan	0x4	Indicates compatibility with Vulkan [5]
Metal	0x8	Indicates compatibility with Metal [1]

11.3 AssetRegistry

The AssetRegistry is an array of data structures which size is defined by the AssetCount field present in the TypeExt.
Below is the data structure to expect as entry in the array:

Name	Type	Size	Notes
VirtualPath	Unsigned	32	Pointer in the Strings section
FileId	Unsigned	32	File section index



All indexes are given as positions in the Section Header Table.

11.3.1 VirtualPath

Asset virtual path as a pointer to a null terminated string in the strings section.

11.4 Strings

The strings section contains a list of null-terminated strings to be referenced by start offset from other sections.

References

[1] Apple. Low level hardware accelerated rendering api. https://developer.apple.com/metal/
Accessed: 2019-12-1.

[2] CGL. Map maps. https://cgl.ethz.ch/teaching/former/vc_master_06/Downloads/Mipmaps_1.pdf/
Accessed: 2019-12-1.

[3] PERPETUAL ENIGMA. Homogeneous coordinate system. https://https://prateekvjoshi.com/2014/06/13/the-concept-of-homogeneous-coordinates/
Accessed: 2019-12-5.

[4] KHRONOS Group. Low level hardware accelerated rendering api. https://www.opengl.org/
Accessed: 2019-12-1.

[5] KHRONOS Group. Low level hardware accelerated rendering api. https://www.khronos.org/vulkan/
Accessed: 2019-12-1.

[6] Jean loup Gailly and Mark Adler. zlib compression algorithm and library. https://www.zlib.net/
Accessed: 2019-12-7.

[7] Microsoft. Low level hardware accelerated rendering api. https://docs.microsoft.com/en-us/windows/win32/directx/
Accessed: 2019-12-1.

[8] Microsoft. Maximum size of a constant buffer available to a shader. https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-vssetconstantbuffers
Accessed: 2019-12-5.

[9] Wikipedia. Graphical processing unit. https://en.wikipedia.org/wiki/Graphics_processing_unit
Accessed: 2019-12-1.

[10] wiseGEEK. Digital rights management. https://www.wisegeek.com/what-is-drm.htm/
Accessed: 2019-12-1.