

Extended Shape Buffer Format

February 9, 2012

Introduction

The shape buffer has been extended to allow the presence of additional modifiers in a shape record. Additional modifiers currently include information about curved segments and point IDs, both of which are explained later. The format extension has been structured in a way that should allow even more modifiers to be added in the future with minimal complication.

The multipatch shape type is also extended to accommodate additional modifiers. This type is used to represent certain 3D geometries with material properties.

Existing shape formats

As a brief review, the current shape structure for polylines, polygons, and multipatches is as follows:

```
long type;
WKSEnvelope boundingBox;1
long cParts,
    cPoints,
    parts[cParts];
    partTypes[cParts]; // multipatches only
WKSPoint points[cPoints];
```

This persistent structure can be read into memory very quickly and converted into a format usable by the Win32 drawing API. These properties need to be preserved.

Polylines, polygons, and multipatches can also be tagged with m and z information. This information appears immediately after the structure shown above. The z information is structured like this:

```
double ZMin;
double ZMax;
double Zs[cPoints];
```

If m-values are present, they appear after the z-values, if present, or after the points if there are no z-values. Their structure is analogous to the z structure.

Overview of the new format

The new format can be easily broken down into sections, some of which are present in all records and others of which are present in only a fraction of records. This section of the document will cover the layout of the record sections and provide a brief description of each. The following sections of the document will then provide more detailed information about each record section.

The first section stores the structural geometry of the shape. It is the most important of the sections and occurs in every shape record. One item stored in the first section is the shape type, which allows the reader to determine which of the other sections will be present.

¹ WKSEnvelope is defined as
struct WKSEnvelope
{
 double xmin, ymin, xmax, ymax;
};

WKSPoint is defined as
struct WKSPoint
{
 double x, y;
};

The other sections contain information about modifiers to the structural geometry. Each section is present only if the modifier associated with it is present in the shape, the second section contains z-data; the third contains m-data; the fourth contains curve data; and the fifth contains point ID data.

Structural geometry section

This section looks nearly identical to an older shape record with no m- or z-values. For points:

```
long    type;
double  x,
        y;
```

For multipoints:

```
long          type;
WKSEnvelope   boundingBox;
long          cPoints;
WKSPoint      points[cPoints];
```

For polygons and polylines:

```
long          type;
WKEnvelope    boundingBox;
long          cParts,
              cPoints,
              parts[cParts];
WKSPoint      points[cPoints];
```

For multipatches:

```
long          type;
WKEnvelope    boundingBox;
long          cParts,
              cPoints,
              parts[cParts];
              partDescriptors[cParts];
WKSPoint      points[cPoints];
```

The difference is that there are some new values possible for the shape type (the first four bytes). The shape type may have any of the values that have been used in the past, in which case the record's format will be unchanged from that previously encountered with the shape type.

Alternatively, the shape type may be a new value. Each new value has two distinct parts. The first part tells what general structure the shape has. It may be any of the following values:

```
esriShapeGeneralPolyline    = 50,
esriShapeGeneralPolygon     = 51,
esriShapeGeneralPoint       = 52,
esriShapeGeneralMultipoint  = 53,
esriShapeGeneralMultiPatch  = 54
```

Shapes with any of these values for the first part of their shape type will hereafter be referred to as having a general type. A special mask, `esriShapeBasicTypeMask`, has been provided for extracting the first part of any shape type. Performing a bitwise AND operation between a shape type and this mask will give the first part of the shape type:

```
structuralPart = shapeType & esriShapeBasicTypeMask;
```

Performing a bitwise AND operation between an older shape type and this mask will return the shape type with no changes.

The second part tells what types of modifiers will be applied to the shape and, consequently, additional sections for which the reader will have to look. The second part may contain any combination of the following bits:

```
esriShapeHasZs      = 0x80000000,
esriShapeHasMs      = 0x40000000,
esriShapeHasCurves = 0x20000000,
esriShapeHasIDs     = 0x10000000,
esriShapeHasNormals = 0x08000000,
esriShapeHasTextures = 0x04000000,
esriShapeHasPartIDs = 0x02000000,
esriShapeHasMaterials = 0x01000000,
```

These modifier bits are never combined with the old shape type values, such as `esriShapePolyline` or `esriShapePolygonM`, since the old values already carry information about their modifiers.

Bit(s)	31	30	29	28	27	26	25	24	23–8	7–0
Purpose	HasZs	HasMs	HasCurves	HasIDs	HasNormals	HasTextures	HasPartIDs	HasMaterials	unused	basic shape type

Additionally, there is a set of values that have special interpretations. If the shape is `esriShapeGeneralPolygon` or `esriShapeGeneralPolyline` but does not have any of the newer modifier bits, it should be interpreted as having curves. To check whether a shape type contains any new modifier bits, perform a bitwise AND operation between the shape type and a special mask provided for this purpose:

```
shapeType & esriShapeNonBasicModifierMask
```

If the result is zero, the shape type does not contain any new modifier bits and should be interpreted as having curves in addition to the modifiers specifically indicated by any other bits that may be present. If the result is nonzero, the shape type contains one of the new bits. In the latter case, the only modifiers present are the ones corresponding to the bits in the shape type.

Part Descriptors data section

This section is present only if the shape type is `esriShapeGeneralMultiPatch`. Multipatch shapes also have the `esriShapeHasPartIDs` bit set.

```
long partDescriptors[cParts];
```

To define and interpret `partDescriptors` values, the following structure should be used:

```
struct esriPartDescriptor
{
    union
    {
```

```

long descriptor; // generalized part ID
struct
{
    int      PartType      : 4;
    unsigned LevelOfDetail : 6;
    int      Priority       : 6;
    unsigned Material       : 16;
};
};
};

```

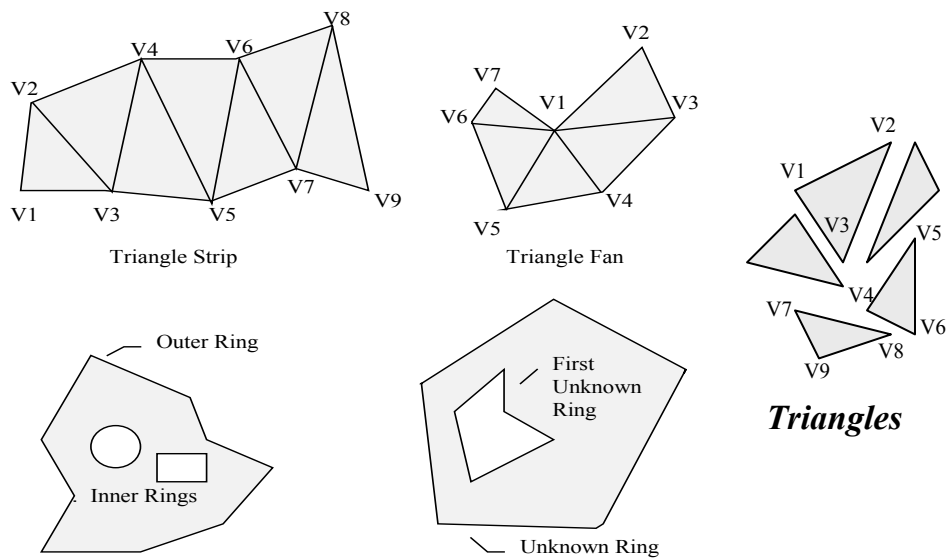
The values used for encoding part type are defined as follows:

```

esriPatchTypeTriangleStrip = 0,
esriPatchTypeTriangleFan   = 1,
esriPatchTypeOuterRing     = 2,
esriPatchTypeInnerRing     = 3,
esriPatchTypeFirstRing     = 4,
esriPatchTypeRing          = 5,
esriPatchTypeTriangles     = 6 // new in esriShapeGeneralMultiPatch

```

Figure 1
Multipatch General Type Part Examples



This figure shows examples of all types of multipatch parts, including the new `esriPatchTypeTriangles`.

Z-data section

The z-data section is present only if the shape has z-values. All the following types indicate the presence of z-values:

esriShapePointZM	= 11,
esriShapePointZ	= 9,
esriShapeMultipointZM	= 18,
esriShapeMultipointZ	= 20,
esriShapePolylineZM	= 13,
esriShapePolylineZ	= 10,
esriShapePolygonZM	= 15,
esriShapePolygonZ	= 19

Z-values will also be present if there is a general type with the `esriShapeHasZs` bit.

For all storage types except points, the z-data section starts by giving the minimum and maximum z-values found in the shape. Next come the z-values found at each point in the shape. Thus, for points:

```
double Z;
```

For multipoints, polygons, and polylines:

```
double ZMin;  
double ZMax;  
double Zs[cPoints];
```

M-data section

The m-data section is present only if the shape has m-values. All the following types indicate the presence of m-values:

esriShapePointM	= 21,
esriShapePointZM	= 11,
esriShapeMultipointM	= 28,
esriShapeMultipointZM	= 18,
esriShapePolylineM	= 23,
esriShapePolylineZM	= 13,
esriShapePolygonM	= 25,
esriShapePolygonZM	= 15

M-values will also be present if there is a general type with the `esriShapeHasMs` bit.

The m-data section is very similar to the z-data section. It gives the minimum and maximum m-values found in the shape, except for points, followed by the m-value found at each point in the shape. For points:

```
double M;
```

For multipoints, polygons, and polylines:

```
double MMin;  
double MMax;  
double Ms[cPoints];
```

Curve data section

The curve section is the first of the sections associated only with general types. All shapes with curves will have a general type, and they will be some sort of polygon or polyline. Shapes with curves will almost always have the `esriShapeHasCurves` bit set, but there is a special case described in the discussion on bit flags in the structural geometry section above. As with the m-data and z-data sections, the curve data section will be present only if the shape has curves. The section begins by giving the number of curved segments in the shape. It then a number of curve subrecords that matches the number of curved segments. Thus, if only one or two segments in a shape are nonlinear, this section will be very short.

```
long cSegmentModifiers;
esriSegmentModifier curvedSegments[cSegmentModifiers];
```

Subrecords vary in length, depending on the type of curve. The general layout follows:

```
struct esriSegmentModifier
{
    long startPointIndex,
        segmentType;
    union {
        SegmentArc          arc;
        SegmentBezierCurve  bezierCurve;
        SegmentEllipticArc  ellipticArc;
        SegmentOther        otherKindsOfSegment;
        ...
    } segmentParams;
}
```

Each subrecord provides the information needed to determine the nonlinear connection between some pair of points j and $j+1$, where j is given by the `startPointIndex` as shown above. The `segmentType` identifies the type of curve described by the subrecord. The remainder of the subrecord is specific to its associated segment type.

The following `segmentType` values are currently defined:

```
enum esriSegmentType
{
    esriSegmentArc = 1,
    esriSegmentLine = 2,
    esriSegmentSpiral = 3,
    esriSegmentBezier3Curve = 4,
    esriSegmentEllipticArc = 5
}
```

The `esriSegmentLine` value will never appear in the curve data section. However, that value should be considered reserved.

Circular arc subrecords

A circular arc is part of a circle. Each endpoint of the segment will lie on a circle, and the segment between them will be one of the two resultant pieces. A circular arc subrecord provides the details about the circle and which of the two pieces should be used:

```
struct SegmentArc
{
    union {
        WKSPoint centerPoint; // If IsPoint is 0. Also, it is ignored if
                                // IsLine is 1.
        double angles[2];      // If IsPoint is 1: start and central angle
    }
}
```

```

        // centerPoint = endPoint
    }

    long Bits;    // Contains, among others, bits for IsPoint, IsLine ...
};

```

Bits contains the following bits (starting with bit 0):

```

IsEmpty;        // Arc is undefined
(ignored)       // The values of these two bits have no effect
(ignored)       // on the shape buffer.
IsCCW;          // 1 if arc is in counterclockwise order.
IsMinor;        // 1 if central angle of arc does not exceed pi.
IsLine;         // Only SP and EP are defined.
IsPoint;        // CP, SP, EP are identical. Angles are stored instead of CP.
DefinedIP;      // IP - interior point. Arcs persisted with 9.2 have
                // endpoints + 1 interior point rather than endpoints and
                // center point so that the arc shape can be recovered after
                // projecting to another spatial reference and back again.
                // Point arcs still replace the center point with SA and CA.

```

Bézier curve subrecords

A Bézier curve segment is simply a section of a third-order Bézier curve (the only type considered here). The curve is defined by four control points. The first and last of these points are given by the segment endpoints found in the structural geometry section of the shape record. This leaves the middle two control points to be stored in the Bézier subrecord:

```

struct SegmentBezierCurve
{
    WKSPPoint controlPoints[2]; // The two middle control points
};

```

Elliptic arc subrecords

An elliptic arc is similar to a circular arc, but instead of being a section of a circle, it is a section of an ellipse. As with a circular arc, the elliptic arc subrecord describes the ellipse on which the segment lies and indicates which of the two parts between the endpoints is to be used:

```

struct SegmentEllipticArc
{
    union {
        WKSPPoint Center; // If CenterTo and CenterFrom are both 0.
        double Vs[2]; // If CenterTo or CenterFrom is 1: from and delta Vs.
                    // Center = fromPoint or toPoint, depending on
                    // values. Vs are similar to angles. If you think of
                    // an ellipse as a stretched circle, then Vs are
                    // angles on the circle being stretched.
    }

    union {
        double Rotation; // If CenterFrom is 1 or CenterTo is 1 or IsLine
                        // is 0. Rotation of the semimajor axis relative to
                        // the x-axis, in radians.
        double FromV; // If CenterTo is 0, CenterFrom is 0, and IsLine
                    // is 1.
    }
};

```

```

double SemiMajor; // On the embedded ellipse, this is the distance
                  // from the center to the point farthest from the
                  // center.

union {
    double MinorMajorRatio; // If CenterFrom is 1 or CenterTo is 1
                            // or IsLine is 0. The ratio between the
                            // semiminor and semimajor axes.
    double DeltaV;          // If CenterFrom is 0 and CenterTo is 0 and
                            // IsLine is 1.
}

long Bits;
};

```

Bits contains the following bits (starting with bit 0):

```

IsEmpty    // 1 if the arc is undefined.
(ignored)  // The values of these five bits have no
(ignored)  // effect on the shape buffer.
(ignored)
(ignored)
(ignored)
IsLine     // 1 if the MinorMajorRatio is 0.
IsPoint    // 1 if the SemiMajor axis is 0.
IsCircular // 1 if the MinorMajorRatio is 1.
CenterTo   // 1 if the Center and the ToPoint are identical.
CenterFrom // 1 if the Center and the FromPoint are identical.
IsCCW      // 1 if the arc is in counterclockwise order.
IsMinor    // 1 if DeltaV does not exceed pi (or go below -pi).
IsComplete // 1 if DeltaV is plus or minus 2 * pi.

```

Point ID data section

As with previous sections, the point ID section will be present only if the shape has point IDs. Only shapes with general types and the `esriShapeHasIDs` bit set will have point IDs.

Minimum and maximum values are not stored for point IDs, so this section consists of just the point ID values found at each point. This also means that there is no difference in the format of this section between points and other geometries (for points, `cPoints` is assumed to be 1).

```
long  PointIDs[cPoints];
```

Point normal data section

As with previous sections, the point normal section will be present only if the shape has point normals. Only shapes with general types and the `esriShapeHasNormals` bit set will have point normals. Face vertex normals are used in shading intensity calculations under illumination.

This section consists of just the point normal component values found at each point. It is recommended that individual normal vectors be normalized—that is, have a magnitude of 1.

```
long  cNormals;          // cNormals = cPoints or 0 if no Normals
float PtNormals[cNormals][3]; // present if cNormals > 0.
```

Point texture coordinates data section

The point texture coordinates section will be present only if the shape has point texture coordinates. Only shapes with general types and the `esriShapeHasTextures` bit set will have point texture coordinates. Face vertex texture coordinates are used in texturing faces when materials with texture data are present.

Texture coordinates can have one, two, or three components, corresponding to the one-, two- or three-dimensional texture image recorded in the materials data section below. Not all shape parts need to have texture coordinates:

```
long cTexPts; // 0 if no texture coordinates present.
long texDim; // 1, 2, or 3 (present if cTexPts > 0).
long texParts[cParts]; // present only if cTexPts > 0.
float texPoints[cTexturePts][texDim]; // present only if cTexPts > 0.
```

An array element `texParts[i]` holds the index of the first texture coordinate within `texPoints[]` which corresponds to part `i`. If part `i` has no texture coordinates, then `texParts[i] = texParts[i+1]`. As a consequence, it is permissible to have `cTexPts <= cPoints`.

Materials data section

As with previous sections, the materials section will be present only if the shape has materials. Only shapes with general types and the `esriShapeHasMaterials` bit set will have materials. Materials are used in rendering faces of an areal shape, typically a multipatch.

```
long cMaterials; // 0 if no materials present.
long texCompressionType; // present only if cMaterials > 0.
long materials[cMaterials + 1]; // present only if cMaterials > 0.
Byte materialBlocks[materials[cMaterials]]; // present only if cMaterials > 0.
```

Where the values used for encoding `texCompressionType` are defined as follows:

```
esriTextureCompressionNever = 1,
esriTextureCompressionNone = 2,
esriTextureCompressionJPEG = 3,
esriTextureCompressionJPEGPlus = 4 // with transparency mask
```

Array element `materials[i]` represents the offset into the `materialBlocks` buffer corresponding to the *i*th material block. The size in bytes of block *i* is consequently equal to `materials[i + 1] - materials[i]`.

`materialBlocks` is the serialized buffer of all material blocks in the range `[0, cMaterials)`.

Each material block has one or more material properties identified by `materialType`:

```
enum materialType
{
    materialColor = 1,
    materialTextureMap = 2,
    materialTransparency = 3,
    materialShininess = 4,
    materialSharedTexture = 5,
    materialCullBackFaces = 6,
    materialEdgeColor = 9,
    materialEdgeWidth = 10,
    materialLast = 11
};
```

Material Properties by materialType:

```
materialColor
{
    materialType type;    // materialColor
    Byte red, green, blue; // [0, 255]
}

materialTextureMap
{
    materialType type; // materialTextureMap
    Byte bpp;          // bytes per pixel
    long width;         // texture width
    long height;        // texture height
    long size;          // texture buffer size
    esriTextureCompressionType tcType;
    Byte texBuff[size]; // texBuff[height][width][bpp] if not compressed
}
```

Note that texBuff is stored row-wise, where:

```
texBuff[0][anyCol] corresponds to texture coordinates (s, t = 0.0)
texBuff[height-1][anyCol] corresponds to texture coordinates (s, t = 1.0)
texBuff[anyRow][0] corresponds to texture coordinates (s = 0.0, t)
texBuff[anyRow][width-1] corresponds to texture coordinates (s = 1.0, t)
```

Color components, if not compressed, are stored in RGBA order. For example, in the case of 3 bytes per pixel (bpp = 3), the components are stored as:

```
texBuff[row][col][0] representing Red value;
texBuff[row][col][1] representing Green value;
texBuff[row][col][2] representing Blue value.
```

If texture is compressed, texBuff contains the compressed stream, and size is the compressed size in bytes.

```
materialTransparency
{
    materialType type; // materialTransparency
    Byte transparency; // percent transparency [0, 100]
}

materialShininess
{
    materialType type; // materialShininess
    Byte shininess;    // percent shininess [0, 100]
}

materialSharedTexture
{
    materialType type; // materialSharedTexture
    long materialIndex; // material with the full materialTextureMap
}

materialCullBackFaces
{
    materialType type; // materialCullBackFaces
}
```

```

materialEdgeColor
{
    materialType type;      // materialEdgeColor
    Byte red, green, blue; // [0, 255]
}

materialEdgeWidth
{
    materialType type; // materialEdgeWidth
    Byte width;        // <= 255
}

```

Summary of complete records

Putting together all the sections described above, one can assemble the format for any type of geometry. They are provided here for ease of reference.

Before the record layout for each shape type, there is a description of how to use the shape type to determine whether the record is of the format in question. The layouts will refer to whether a shape has certain modifiers. Refer to the section about any given modifier to find out how to determine whether the shape has that modifier.

A shape is a point if any of the following is true:

```

shapeType == esriShapePoint
shapeType == esriShapePointZ
shapeType == esriShapePointM
shapeType == esriShapePointZM
(shapeType & esriShapeBasicTypeMask) == esriShapeGeneralPoint

```

Table 1
Point Record Contents

Position	Field	Type	Quantity	Byte Order
Byte 0	ShapeType	Integer	1	Little
Byte 4	X	Double	1	Little
Byte 12	Y	Double	1	Little
Byte 20 ²	Z	Double	1	Little
Byte A ³	M	Double	1	Little
Byte B ⁴	ID	Integer	1	Little

² Present only if shape has z-values;

³ Present only if shape has m-values; A = 20 + (0 if no z-values, 8 if z-values present)

⁴ Present only if shape has IDS; B = A + (0 if no m-values, 8 if m-values present)

A shape is a multipoint if any one of the following is true:

```

shapeType == esriShapeMultipoint
shapeType == esriShapeMultipointZ
shapeType == esriShapeMultipointM
shapeType == esriShapeMultipointZM
(shapeType & esriShapeBasicTypeMask) == esriShapeGeneralMultipoint

```

Table 2
Multipoint Record Contents

Position	Field	Type	Quantity	Byte Order
Byte 0	ShapeType	Integer	1	Little
Byte 4	Box	Double	4	Little
Byte 36	NumPoints	Integer	1	Little
Byte 40	Points	Point	NumPoints	Little
Byte C ⁵	ZMin	Double	1	Little
Byte C + 8 ⁵	ZMax	Double	1	Little
Byte C + 16 ⁵	Zs	Double	NumPoints	Little
Byte D ⁶	MMin	Double	1	Little
Byte D + 8 ⁶	MMax	Double	1	Little
Byte D + 16 ⁶	Ms	Double	NumPoints	Little
Byte E ⁷	IDs	Integer	NumPoints	Little

⁵ Present only if shape has z-values; C = 40 + (16 * NumPoints)

⁶ Present only if shape has m-values; D = C + [0 if no z-values, 16 + (8 * NumPoints) if z-values present]

⁷ Present only if shape has IDs; E = D + [0 if no m-values, 16 + (8 * NumPoints) if m-values present]

A shape is a polyline if any one of the following is true:

```
shapeType == esriShapePolyline
shapeType == esriShapePolylineZ
shapeType == esriShapePolylineM
shapeType == esriShapePolylineZM
(shapeType & esriShapeBasicTypeMask) == esriShapeGeneralPolyline
```

A shape is a polygon if any one of the following is true:

```
shapeType == esriShapePolygon
shapeType == esriShapePolygonZ
shapeType == esriShapePolygonM
shapeType == esriShapePolygonZM
(shapeType & esriShapeBasicTypeMask) == esriShapeGeneralPolygon
```

Table 3
Polyline and Polygon Record Contents

Position	Field	Type	Quantity	Byte Order
Byte 0	ShapeType	Integer	1	Little
Byte 4	Box	Double	4	Little
Byte 36	NumParts	Integer	1	Little
Byte 40	NumPoints	Integer	1	Little
Byte 44	Parts	Integer	NumParts	Little
Byte F ⁸	Points	Point	NumPoints	Little
Byte G ⁹	ZMin	Double	1	Little
Byte G + 8 ⁹	ZMax	Double	1	Little
Byte G + 16 ⁹	Zs	Double	NumPoints	Little
Byte H ¹⁰	MMin	Double	1	Little
Byte H + 8 ¹⁰	MMax	Double	1	Little
Byte H + 16 ¹⁰	Ms	Double	NumPoints	Little
Byte I ¹¹	NumCurves	Integer	1	Little
Byte I + 4 ¹¹	SegmentModifiers	SegmentModifier	NumCurves	Little
Byte J ¹²	IDs	Integer	NumPoints	Little

⁸ F = 44 + (4 * NumParts)

⁹ Present only if shape has z-values; G = F + (16 * NumPoints)

¹⁰ Present only if shape has m-values; H = G + [0 if no z-values, 16 + (8 * NumPoints) if z-values present]

¹¹ Present only if shape has curves; I = H + [0 if no m-values, 16 + (8 * NumPoints) if m-values present]

¹² Present only if shape has IDs; J = I + [0 if no curves, 4 + (28 per circular arc) + (40 per Bézier curve) + (52 per elliptic arc) if curves present]

A shape is a multipatch if any one of the following is true:

```
shapeType == esriShapeMultiPatch
shapeType == esriShapeMultiPatchM
(shapeType & esriShapeBasicTypeMask) == esriShapeGeneralMultiPatch
```

Table 4
Multipatch General Type Record Contents

Position	Field	Type	Quantity	Byte Order
Byte 0	ShapeType	Integer	1	Little
Byte 4	Box	Double	4	Little
Byte 36	NumParts	Integer	1	Little
Byte 40	NumPoints	Integer	1	Little
Byte 44	Parts	Integer	NumParts	Little
Byte K ¹³	PartDescriptors	Integer	NumParts	Little
Byte L ¹⁴	Points	Point	NumPoints	Little
Byte M ¹⁵	ZMin	Double	1	Little
Byte M + 8 ¹⁵	ZMin	Double	1	Little
Byte M + 16 ¹⁵	Zs	Double	NumPoints	Little
Byte N ¹⁶	NumMs	Integer	1	Little
Byte N + 4 ¹⁷	MMin	Double	1	Little
Byte N + 12 ¹⁷	MMax	Double	1	Little
Byte N + 20 ¹⁷	Ms	Double	NumMs	Little
Byte O ¹⁸	NumIds	Integer	1	Little
Byte O + 4 ¹⁷	IDs	Integer	NumIds	Little
Byte P ¹⁹	NumNormals	Integer	1	Little
Byte P + 4 ¹⁷	Normals	Float[3]	NumNormals	Little
Byte Q ²⁰	NumTex	Integer	1	Little
Byte Q + 4 ¹⁷	TexDim	Integer	1	Little
Byte Q + 8 ¹⁷	TexParts	Integer	NumParts	Little
Byte Q + 8 + (4 * NumParts) ¹⁷	TexCoords	Float[TexDim]	NumTex	Little
Byte R ²¹	NumMaterials	Integer	1	Little
Byte R + 4 ¹⁷	TexCompType	Integer	1	Little
Byte R + 8 ¹⁷	Materials	Integer	NumMaterials + 1	Little
Byte S ²²	Material	Material block	NumMaterials	Little

¹³ K = 44 + (4 * NumParts)

¹⁴ L = K + (4 * NumParts)

¹⁵ M = L + (16 * NumPoints)

¹⁶ N = M + 16 + (8 * NumPoints)

¹⁷ Present only if shape has the corresponding modifier

¹⁸ O = N + [4 if no m-values, 20 + (8 * NumMs) if m-values present]

¹⁹ P = O + [4 if no IDs, (4 + 4 * NumIds) if IDs present]

²⁰ Q = P + [4 if no normals, (4 + 12 * NumNormals) if normals present]

²¹ R = Q + [4 if no texture coordinates, 8 + (4 * NumParts) + (4 * TexDim * NumTex) if texture coordinates present]

²² Present only if shape has materials; S = R + (12 + 4 * NumMaterials)