Extended Shape Buffer Format

June 20, 2012

Introduction

The shapefile record structure has been extended to allow for the presence of additional modifiers in a shape record. Additional modifiers currently include information about curved segments and point IDs, both 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.

Multipatch shape type is also extended to accommodate some 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 this:

```
long type;
WKEnvelope 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 very quickly converted into a format useable 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 Ms are present, they appear after the Zs if Zs are present, or after the points if there are no Zs. 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 only in 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 for the shape. The third section contains M data for the shape. The fourth section contains curve data for the shape. The fifth section contains point ID data for the shape.

Structural geometry section

This section looks nearly identical to an older shape record with no Ms or Zs. 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 OR between a shape type and this mask will give the first part of the shape type:

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

Performing a bitwise OR 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 which additional sections the reader will have to look for. 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	has Zs	has Ms	has Curves	has IDs	has Normals	has Textures	has PartIDs	has Materials	unused	basic shape type

Additionally, there is a set of values that have special interpretations: If the shape is esriShapeGeneralPolygon or esriShapeGeneralPolyline, but it 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, one can perform a bitwise OR 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 non-zero, 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 only present if the shape type is esriShapeGeneralMultiPatch. Multipatch shapes also have the esriShapeHasPartIDs bit set.

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

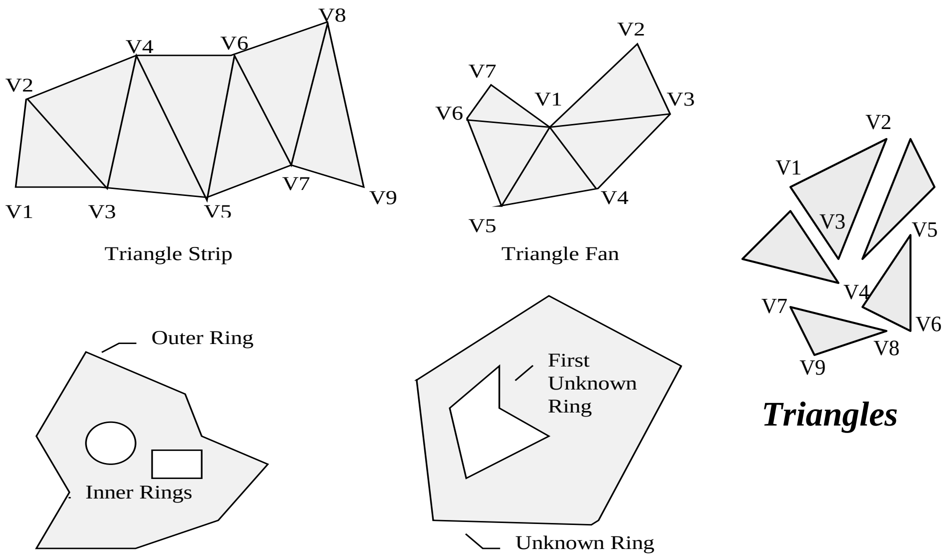
To define and interpret `partDescriptors` values, the following struct should be used:

```
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



Examples of all types of Multipatch parts, including the new esriPatchTypeTriangles.

`LevelOfDetail` value is a hint reserved for future use.

`Priority` is an integer value in the range of [-32,31] indicating the relative priority between the coincident surface patches. The default is zero as the most typical value.

`Material` value is the index into `materials[]` array (see **Materials data** section bellow).

Z data section

This section is only present if the shape has Zs. All of 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
```

Zs will also be present if there is a general type with the esriShapeHasZs bit.

For all storage types except points, the Z section starts by giving the minimum and maximum Z values found in the shape. Next come the Z value 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

This section is only present if the shape has Ms. All of 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,
```

Ms will also be present if there is a general type with the esriShapeHasMs bit.

The M section is very similar to the Z 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 structural geometry section above, in the discussion of bit flags. As with the M and Z sections, the curve section will only be present if the shape has curves. The section begins by giving the number of curved segments in the shape. It then provides that number of curve sub-records. Thus if only one or two segments in a shape are non-linear, this section will be very short.

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

These sub-records 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 sub-record provides the information needed to determine the non-linear 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 sub-record. The remainder of the sub-record 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 sub-records

A circular arc is a part of a circle. Each end point of the segment will lie on a circle, and the segment between them will be one of the two resulting pieces. A circular arc sub-record provides the details about the circle and about 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
(reserved)
(reserved)
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 persist
              // 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
```

Bezier curve sub-records

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

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

Elliptic arc sub-records

An elliptic arc is similar to a circular arc. Instead of being a section of a circle, though, it is a section of an ellipse. As with a circular arc, the elliptic arc sub-record describes the ellipse on which the segment lies, and indicates which of the two parts between the end points is to be used:

```
struct SegmentEllipticArc
{
  union {
    WKSPoint 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 furthest 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
(reserved)
(reserved) // These 5 bits are used only at run-time. Their values
(reserved) // are ignored when reading from disk.
(reserved)
(reserved)
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 only be present 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 only be present 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 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 only be present 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.

The Texture Coordinates can have 1, 2, or 3 components, corresponding to 1-, 2-, or 3-dimensional texture image recorded in Materials data section bellow. 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 only be present 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, starting at 4, into `materialBlocks` buffer corresponding to the Material `i` block. The size in bytes of the Material block `i` is consequently equal to `materials[i+1]-materials[i]`.

The first 4 bytes of the `materialBlocks` are reserved for internal use (e.g.: for storing the uncompressed buffer size), followed by the serialized buffer of all MaterialBlocks in the range `[0, cMaterials-1]`. Note again that `materials[0]`cannot be less than 4 as a consequence.

MaterialBlock has one or more material properties identified by `materialType` value stored in the first byte.

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

  materialTextureMap:
  {
```

```
  Byte type;          // materialTextureMap
  Byte bpp;           // Bytes per pixel
  short width;        // texture width
  short height;       // texture height
  long size;          // texture buffer size
  long tcType;        // esriTextureCompressionType
  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 **R**ed value;
`texBuff[row][col][1]` representing **G**reen value;
`texBuff[row][col][2]` representing **B**lue value.

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

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

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

materialSharedTexture:
```
{
  Byte type;          // materialSharedTexture
  long materialIndex; // material with the full materialTextureMap
}
```

materialCullBackFaces:
```
{
  Byte type; // materialCullBackFaces
}
```

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

materialEdgeWidth:
```
{
  Byte type;  // materialEdgeWidth
  Byte width; // <= 255
}
```

Summary of complete records

Putting together all of 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 or not a shape has certain modifiers. Refer back to the section on any given modifier to find out how to determine if the shape has that modifier.

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

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

Table 1
Point Record Contents

Position	Field	Type	Number	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

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	Number	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

[2] Only present if shape has Zs.
[3] Only present if shape has Ms. A = 20 + (0 if no Zs OR 8 if Zs present)
[4] Only present if shape has IDS. B = A + (0 if no Ms OR 8 if Ms present)
[5] Only present if shape has Zs. C = 40 + (16 * NumPoints)

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

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	Number	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

[6] Only present if shape has Ms. D = C + [0 if no Zs OR 16 + (8 * NumPoints) if Zs present]

[7] Only present if shape has IDs. E = D + [0 if no Ms OR 16 + (8 * NumPoints) if Ms present]

[8] F = 44 + (4 * NumParts)

[9] Only present if shape has Zs. G = F + (16 * NumPoints)

[10] Only present if shape has Ms. H = G + [0 if no Zs OR 16 + (8 * NumPoints) if Zs present]

[11] Only present if shape has curves. I = H + [0 if no Ms OR 16 + (8 * NumPoints) if Ms present]

[12] Only present if shape has IDs. J = I + [0 if no curves OR 4 + (28 per circular arc) + (40 per bezier curve) + (52 per elliptic arc)]

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	Number	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

Note that NumMs, NumIds, and NumNormals can only have two values: 0 or NumPoints, depending on the presence of the corresponding modifier attribute.

[13] K = 44 + (4 * NumParts)

[14] L = K + (4 * NumParts)

[15] M = L + (16 * NumPoints)

[16] N = M + 16 + (8 * NumPoints)

[17] Only present if shape has the corresponding modifier.

[18] O = N + [4 if no Ms OR 20 + (8 * NumMs) if Ms present]

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

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

[21] R = Q + [4 if no Texture Coords OR 8 + (4 * NumParts) + (4 * TexDim * NumTex) if Texture Coords present]

[22] Only present if shape has Materials. S = R + (12 + 4 * NumMaterials).