# Visible Scalable Terrain (ViSTa) Format

Marsette Vona, Mark Powell

Version 0.8 March 27, 2002

## 1 Introduction

Visible Scalable Terrain (*ViSTa*) is a format for the production, interchange, and display of 3D terrain data that is specifically suited to stereo vision based robotic applications. ViSTa is designed specifically to support both data scalability as well as visualization accuracy, performance, and optimal resource usage.

The geometry of a vista terrain[1] consists of a set of zero or more triangles and a set of zero or more isolated points. The appearance of a vista terrain is texture-mapped from the images originally acquired by the vision hardware.

### 1.1 Visibility

A vista terrain specifies an association for each point and triangle to a specific texture map image, such that

- the texture map image is the rectified version of an image acquired by the left camera of the stereo vision hardware
- the respective point or triangle is *visible* in the texture image, where the definition of *visible* is that
  1. the point or triangle is not occluded in any part in the texture image by any triangle in the same LOD in the vista terrain
  2. the geometric back-projection of the point or front face of the triangle into the image plane is entirely within the bounds of the image

Different points and triangles in a single ViSTa file may be associated with different texture images.

[1] We are employing the term "vista" as a name in this usage, not an acronym, so the phrase "vista terrain" is not redundant.

### 1.2 Scalability

A single ViSTa file can contain terrains ranging from a fraction of the data from a single stereo pair to a full panorama of stereo pairs or more.

A vista terrain contains a pool of vertices and a set of texture map references. The geometry is specified as a set of Levels Of Detail (LOD) to support visualization performance and optimal resource usage. Each LOD is defined as a set of patches. Each patch defines either set of triangles or a set of isolated points, all of which are associated with a specified texture map image for that patch.

Practically, systems which read vista terrains **may** refuse to accept extremely large ViSTa files. In no event may a single ViSTa file be larger than $2^{31} - 1$ bytes (2 GB).

Another practical consideration is that, since ViSTa uses 32-bit floating point values to represent geometric data, there is a trade-off between the geometric extent of a vista terrain and the representable resolution (see Section 5 for details).

### 1.3 Document Structure

The next two subsections provide some background on the motivations for the development of ViSTa. Sections 4 through 7 describe semantic details of ViSTa, and sections 8 and 9 give the syntactic details of ViSTa.

## 2 Design Goals

The design goals of ViSTa are as follows:

- data accuracy, especially with respect to the abilities to display and pick accurate locations on the terrain and to accurately display overlaid representations of auxiliary science data
- minimization of file size
- platform portability
- run-time efficiency in computation and memory usage for systems which create, process, and display vista terrains
- support of both high-resolution/resource-intensive and lower-resolution/-less-resource-intensive (e.g. public outreach) applications
- support of both single-wedge (i.e. all vertices visible from the left image of a single stereo pair) and merged multi-wedge terrains
- ease of generation from existing stereo vision data
- ease of integration with systems which handle terrain (e.g. SAP, SUMMITT, RSVP)

## 3 Relation to ASD

ASD is an SGI *Performer* scheme for representing texture-mapped triangle meshes with multiple LOD. ASD is an API and not a file format, however Jack Morrison proposed a format suitable for storing ASD data in [1], and ASD data can also be written to files in Performer Binary format (PFB).

ViSTa is designed to be relatively easy to generate by systems which use ASD for the following reasons:

- systems like SUMMITT and RSVP already use ASD
- the ASD API is documented reasonably well in [3]
- the capabilities of ASD are a reasonably good superset of the required capabilities for ViSTa

However, it may be inconvenient to efficiently load a vista terrain into an ASD-based system. Some features of ASD are not required for ViSTa and are specifically omitted in order to reduce storage size and processing requirements. Differences between ViSTa and ASD include the following:

- ViSTa does not store any LOD difference (morphing) information
- ViSTa does not store information about surface normals or vertex colors
- ViSTa does not include vertex attribute fields that are separate from vertex coordinate fields
- ViSTa only stores information about triangle strips, not individual triangles (and the representation of triangle strips in ViSTa is not the same as in ASD)
- ViSTa requires that all triangles in a strip have the same texture map
- ViSTa allows isolated points in addition to triangles
- ViSTa introduces the concept of triangle strip and point cloud *patches* and requires that no two triangle strips within a single triangle strip patch and no two points in a point cloud patch have different texture maps
- ViSTa does not allow the use of automatic texture coordinate generation — texture map coordinates must be explicitly specified for all vertices

# 4 Some Details Are Implementation-Dependent

Some details are left as implementation-dependent; these will be explicitly called out below. Each ViSTa implementation is defined by a separate document which specifies how each of the implementation-dependent details are to be handled. A ViSTa implementation may be associated closely with a specific system that produces and/or displays ViSTa data, but this is not required. It is permissible for a producer or displayer of ViSTa data to support multiple ViSTa implementations.

Each specific implementation is assigned a 4-byte implementation identifier (which is called out in the documentation for that implementation). A system which creates a ViSTa file **must** specify the implementation it is using by writing the corresponding implementation identifier in the header section of the file (see Section 10.1). Systems which display vista terrains **must** check this field and interpret the data according to the indicated implementation, or display an error message if they do not support the implementation.

# 5 Coordinate Frame and Units

All spatial geometric data in a vista terrain are specified in units of meters.

All vertices in a vista terrain are specified in the same coordinate system. The specification of this coordinate system is implementation-dependent, but in all cases must be Cartesian and right-handed. A fixed space is reserved in every ViSTa file for implementation-dependent specification of coordinate system (this space may be unused in some implementations, e.g. if the coordinate system in that implementation is implicit).

Since 32-bit floating point values are used to represent all geometric data in ViSTa (see Section 8.1), there is a trade-off between the geometric extent of a vista terrain and the representable resolution. For example, to maintain 1mm representable resolution, the maximum extent of a vista terrain can not be more than about 32km. For this reason, a ViSTa implementation may optionally specify limits on the maximum representable geometric extent, though in many applications this is unnecessary because the terrain extents are already limited by design.

It is also possible for ViSTa implementations to permit large datasets to be split into multiple separate vista terrains, each including high–resolution locale data in an implementation-dependent format in their `CoordinateSystem` fields (see Section 10.4).

# 6 Levels of Detail

At least one, and possibly more, ordered LOD are specified in a single ViSTa file. Multiple LODs within a single ViSTa file are presented in increasing order from least-detailed to most-detailed. Each LOD is implicitly assigned a non-negative integer index according to its position in the file. The first LOD in the file is assigned index 0, and subsequent LODs are assigned sequential indices.

Unless specifically directed otherwise, vista terrains produced by one system for interchange with another system **must** include the most detailed representation of the terrain that the originating system can provide as the most detailed (i.e. last) ViSTa LOD. Additional LOD with progressively less detail **should** also be included if available and appropriate. If the originating system has good reason to choose the resolution of these less-detailed LOD specifically then it **should** do that. Otherwise, guidelines for generating the less-detailed LOD are that LOD $(i+1)$ **should** have about twice as many vertices as LOD $i$, and LOD 0 **should** have on the order of 100 to 1000 vertices.

For compact representation, systems which generate vista terrains **should** make a best-effort to re-use vertices from less-detailed LOD in more-detailed LOD. Specifically, let the set of vertices of LOD $i$ be $V_i$, and let the set of vertices of LOD $i+1$ be $V_{i+1}$. Then $V_{i+1}$ is separable into two disjoint sets: $V_{i+1} = V_{i+1}^{\text{old}} \cup V_{i+1}^{\text{new}}$, where $V_{i+1}^{\text{old}} \subseteq V_i$ and $V_{i+1}^{\text{new}} \cap V_i = \emptyset$. LOD $i$ and $i+1$ **should** be constructed to maximize the size of $V_{i+1}^{\text{old}}$.

The concept of LOD in ViSTa is restricted to the geometry of the terrain. Specifically, the LOD information in a vista terrain does *not* have any relation to the texture map(s) associated with the terrain. Each LOD in a vista terrain must reference exactly the same set of texture map images as every other LOD in the terrain.

However, systems which visualize vista terrains **may** implement some form of texture LOD switching on their own.

# 7 Texture Mapping and The Relation of Terrain Geometry to Raw Data

Every patch (set of triangles or set of isolated points) in a vista terrain has a single associated texture map image. As described below in Section 10.3, texture maps are identified in an implementation-dependent way as persistent references to the texture image data.

Vista terrains may only refer to texture map images that are the rectified versions of images acquired by the left camera of the stereo hardware (or filtered and/or linearly scaled versions thereof).

Included with the specification of every vertex in a vista terrain is a pair of normalized texture coordinates $(s,t), s \in [0 \ldots (1-1/N)], t \in [0 \ldots (1-1/M)]$, where the associated texture map is $N$ pixels wide and $M$ pixels tall. The origin of the normalized texture coordinate system is the center of the pixel at the *upper left* corner of the texture image, and the orientation is such that $s$ increases to the right in the image and $t$ increases *down*.

Because the texture coordinates are normalized, there are no restrictions on the dimensions of the texture images. However, systems which visualize vista terrains **may** re-scale the texture images as graphics hardware requires.

Unless specifically directed otherwise, vista terrains produced by one system for interchange with another system **must** reference the highest-resolution versions of the associated texture map images, if multiple versions are available. The specific location and mode of transfer (e.g. in a mission database) of the texture map images is implementation-dependent.

## 7.1 Accuracy of Texture Coordinates

The accuracy of the texture coordinates associated with a terrain vertex is important, as it may relate directly to the accuracy of selected and displayed locations on the terrain surface in ViSTa visualization implementations.

Texture coordinates for a vertex are *accurate* if they define a point in the texture image plane inside the pixel containing the geometric back-projection point of the vertex, given the best-known camera calibration and camera pose data[2]. This definition can also be viewed as an algorithm to generate texture coordinates for an arbitrary vertex, provided that a texture image in which the vertex is visible is already known. Additionally, such texture coordinates are available as a trivial by-product of basic stereo reconstruction algorithms which perform the forward mapping of image pixels to 3D vertices: if such an algorithm maps pixel $\mathbf{p} = (i,j)$ of the rectified left image[3] to 3D vertex $\mathbf{v}$, and the rectified left image is $N$ pixels wide and $M$ pixels tall, then the normalized texture coordinates associated with $\mathbf{v}$ are $(i/N, j/M)$.

If a system generating a vista terrain cannot determine a rectified left image in which a specific vertex is visible and/or cannot determine accurate texture coordinates in such an image for that vertex, it **must** not include that vertex in the terrain.

Each triangle and each isolated point in a vista terrain is associated with exactly one texture map image. In the case of a single terrain containing geometry synthesized from multiple stereo pairs, some triangles might not be visible (as defined in Section 1.1) from the left image of any single stereo pair. Such triangles **must** not be included in the vista terrain.

[2] I.e., the best known data at the time that the vista terrain was generated.

[3] With the pixel origin $(0,0)$ in the *upper left* corner of the image, $i$ increasing right, and $j$ increasing *down*.

# 8 Format Preliminaries

ViSTa is a binary format for reduced size, increased compressibility, and increased computational efficiency. ViSTa structures are all defined as multiples of 32-bit words in order to support memory alignment at word boundaries for efficient manipulation on 32-bit architectures.

For storage and transmission purposes, a ViSTa file **must** be given a name ending in ".vst".

## 8.1 Definitions and Data Types

In this document, a *float* is an IEEE754 32-bit floating point number, and an *int* is a 32-bit two's compliment signed integer.

All *float* and *int* in a ViSTa file are stored using the byte order identified in the `VSTHeader` field (Section 10.1). Unless otherwise specified, systems which generate ViSTa files **should** use little-endian (LSB-first, Intel) byte order.

Character constants are written with single quotes as they are in the C language (e.g. 'A') and refer to 8-bit ASCII character codes. The syntax *byte*$[n]$ refers to an ordered array of $n$ *byte*s, with the 0th byte first in the file and the $(n-1)$th byte last.

Fields which are marked *reserved* **must** be set to 0 by systems which generate ViSTa files.

# 9 ViSTa Binary Layout

At the top level, a ViSTa file is an ordered sequence of `fields` which follows this grammar:

ViSTa := `VSTHeader`
`BoundingBox`
`TextureRef`$^T$
`CoordinateSystem`
`Vertex`$^+$
LOD$^+$

LOD := `LODHeader`
`BoundingBox`$^T$
Patch$^+$

Patch := `PatchHeader`
`IndexArrayLength`$^n$
IndexArray$^n$

IndexArray := `Index`$^+$

As long as the maximum file size restriction of $2^{31}-1$ bytes is observed, a vista terrain may contain an unlimited number of `TextureRef`, `Vertex`, and LOD, each LOD may contain an unlimited number of patches, each patch may contain an unlimited number of index arrays, and each index array may contain an unlimited number of indices.

Practically, systems which read ViSTa files **may** refuse to accept extremely large files, even if those files are within the $2^{31}-1$ byte limit.

A vista terrain **must** contain at least one `TextureRef`, one `Vertex`, and one LOD, and it is valid for a vista terrain to contain only one `TextureRef`, one `Vertex`, and one LOD.

## 9.1 Patches

A Patch is either a set of triangles (a triangle strip patch) or a set of isolated points (a point cloud patch) in an LOD, all of which use the same texture map. As described below in Section 10.7, data in the `PatchHeader` determine whether the vertices referenced by a patch are to be interpreted as triangle strips or point clouds. In either case, the vertex data for a Patch is specified as a contiguous set of $n$ arrays of `Index` fields, each of which references a specific `Vertex` field from the array at the beginning of the file. The length of the $i$th `Index` array in a patch is given by the $i$th `IndexArrayLength` field in that patch.

No triangle or point included in one patch in an LOD may be included in any other patch in that LOD.

It is not required that all triangles/points which are textured from the same texture image be members of a single patch within an LOD, i.e. multiple triangle strip/point cloud patches within a single LOD may be associated with a single texture image. However, for compact representation, unless explicitly directed otherwise, systems which generate vista terrains **should** place all triangles in an LOD which share the same texture map image into the same triangle strip patch in that LOD, and similarly all isolated points in an LOD which share the same texture map image **should** be placed in the same point cloud patch in that LOD.

## 9.2 Index Assignment and Vertex Order

All `TextureRef` and `Vertex` fields in a ViSTa file, as well as all LOD in the file, are assigned global non-negative integer indices. Indices are zero-based and are assigned implicitly according to the corresponding object's global position in the file. Separate indices are used for each of the indexed object types. The first instance of an object of a specific type is assigned index 0, and subsequent instances of that type of object throughout the file are assigned sequential indices.

Systems which generate ViSTa files **should** sort the array of `Vertex` in the file so that a `Vertex` that is only referenced by LOD greater than $i$ comes after all `Vertex` fields that are referenced by all LOD less than or equal to $i$.

## 9.3 Triangle Strip Structure

The structure of the triangle strips in a triangle strip Patch is implied from the order in which the vertices composing the strips are referenced. Each `Index` array in the Patch defines a separate triangle strip. The first three `Index` fields in an `Index` array (`Index` fields 0, 1, and 2 in the array) identify the vertices of the 0th triangle in the strip in Counter-ClockWise (CCW) order. The length of the strip, $s_n$, the number of triangles it contains, equals the length of the `Index` array minus two. The $s_n - 1$ `Index` fields which follow the first three of the strip define the remaining triangles in the strip: the vertex identified by the `Index` at position $i > 2$, $V_i$, in the `Index` array is combined with vertices identified by the `Index` fields at positions $i-1$ and $i-2$, $V_{i-1}$ and $V_{i-2}$, to form a triangle with vertices $V_{i-2}$, $V_{i-1}$, and $V_i$, interpreted in CCW order if $i$ is even and ClockWise (CW) order if $i$ is odd.

# 10 ViSTa Fields

## 10.1 VSTHeader

| Type | Value |
|---|---|
| *byte* | 'V' |
| *byte* | 'S' |
| *byte* | 'T' |
| *byte* | '\0' |
| *byte*[4] | Byte order identifier. {3, 2, 1, 0} if the file is encoded with big-endian byte order, {0, 1, 2, 3} if the file is encoded with little-endian byte order. |
| *int* | Binary major version number (0 for ViSTa files that correspond with the version of ViSTa described in this document) |
| *int* | Binary minor version number (8 for ViSTa files that correspond with the version of ViSTa described in this document) |
| *byte*[4] | Implementation Identifier (see Section 4) |
| *byte*[8] | Reserved for future use |
| *int* | Number of `TextureRef` listed in the file ($\geq 1$) |
| *int* | Number of `Vertex` listed in the file ($\geq 1$) |
| *int* | Number of LOD listed in the file ($\geq 1$) |

## 10.2 BoundingBox

| Type | Value |
|---|---|
| *float* | $x_{min}$ (see below) |
| *float* | $y_{min}$ (see below) |
| *float* | $z_{min}$ (see below) |
| *float* | $x_{max}$ (see below) |
| *float* | $y_{max}$ (see below) |
| *float* | $z_{max}$ (see below) |

The `BoundingBox` field is used in two contexts in a ViSTa file. The first context is directly after `VSTHeader` and specifies a bounding box for all geometry defined in all LODs in the file (this is referred to elsewhere as "the terrain bounding box"). All vertices in all LOD specified in the file must be inside this bounding box (or exactly coincident with one or more of its faces).

The other context in which `BoundingBox` fields occur is in an array directly after each `LODHeader`. Such arrays of `BoundingBox` are always exactly as long as the array of `TextureRef`, and specify bounding boxes for the geometry in the LOD that is textured by the corresponding texture image. That is, if a ViSTa file contains $T$ `TextureRef` then every `LODHeader` is followed by exactly $T$ `BoundingBox`, and `BoundingBox` $t$ in the array for LOD $i$ gives a bounding box for the set of all vertices referenced by all patches in LOD $i$ which specify that they are textured by texture $t$.

`BoundingBox` fields always define rectilinear bounding boxes with faces normal to the terrain coordinate system axes, and the bounding box coordinates are always specified in meters in the same coordinate system as the terrain vertices.

Systems which generate ViSTa files **should** generate bounding boxes that are as tight as possible. None of the bounding box parameters can be IEEE754 NaN or ±Infinity.

## 10.3 TextureRef

| Type | Value |
|---|---|
| *byte*[2048] | An implementation-dependent persistent reference to the rectified version of an image acquired by the left camera of the stereo hardware (see below). |

The format of the persistent reference is implementation-dependent, as is the location (e.g. in a mission database) and binary format of the texture map image.

No two `TextureRef` in a vista terrain may reference the same texture.

See also Section 7.

## 10.4 CoordinateSystem

| Type | Value |
|---|---|
| *byte*[4096] | Coordinate system specification (implementation-dependent) |

As mentioned in Section 5, the `CoordinateSystem` field specifies the coordinate frame used by all geometry in the vista terrain in an implementation-dependent way. Some implementations **may** choose not to use this data, and identify the coordinate system in some other way, according to their specific documentation.

## 10.5 Vertex

| Type | Value |
|---|---|
| *float* (LSB first) | texture coordinate $s$ (see Section 7 for numeric limits) |
| *float* (LSB first) | texture coordinate $t$ (see Section 7 for numeric limits) |
| *float* (LSB first) | spatial coordinate $x$ |
| *float* (LSB first) | spatial coordinate $y$ |
| *float* (LSB first) | spatial coordinate $z$ |

No *float* entry in a `Vertex` field may be an IEEE754 NaN or $\pm$Infinity.

The spatial coordinates are given in units of meters. The coordinate frame is implementation-dependent, as describe in Section 5.

Systems which generate ViSTa files **should** sort the array of `Vertex` in the file so that a `Vertex` that is only referenced by LOD greater than $i$ comes after all `Vertex` fields that are referenced by all LOD less than or equal to $i$.

## 10.6 LODHeader

| Type | Value |
|---|---|
| *int* | Total size of this LOD, including `LODHeader` (bytes) |
| *byte*[8] | Reserved for future use |
| *int* | Total number of *distinct* `Vertex` referenced in this LOD ($\geq 1$) |
| *float* | Eyepoint distance to terrain bounding box centroid (as specified directly after `VSTHeader`) threshold below which to consider switching to the next higher-resolution LOD ($\geq 0$; see below) |
| *int* | Number of Patches listed in this LOD ($\geq 1$) |
| *int* | Highest `Vertex` index referenced in this LOD ($\geq 0$) |

The total size of an LOD is the size of the `LODHeader`, plus the sum of the sizes of the `BoundingBox` in the array following the `LODHeader`, plus the sum of the sizes of all the Patches in the LOD.

The LOD switch threshold **may** not be honored by systems which visualize vista terrains. The switch threshold of the last LOD in the file is always ignored, even if the file contains only one LOD. For a terrain with $n \geq 2$ LOD, the switch thresholds of LOD $[0 \ldots (n-2)]$ must be non-negative and monotonically decreasing.

Systems which generate vista terrains **should** set the LOD switch thresholds to the best of their ability so that the terrain will look good when rendered. The following algorithm **may** be used if the generating system has no better way to compute the LOD switch thresholds (contributed by Jack Morrison [2]):

For each LOD $i$:

1. Compute $w_i$, an estimate of the average "width" of a triangle in LOD $i$, according to the following formula

$$w_i = \frac{\text{(average terrain bounding box side)}}{\sqrt{\text{number of triangles in LOD } i}}$$

2. Compute $t_i$, the LOD switch threshold for LOD $i$, according to the following formula, which makes $t_i$ the distance at which a triangle of "width" $w_i$ subtends a viewing angle of 1°:

$$t_i = \frac{0.5 w_i}{\tan 0.5°}$$

   Or, use the approximation:

$$t_i \approx 57 w_i$$

See also Section 6.

## 10.7 PatchHeader

| Type | Value |
|---|---|
| *byte*[8] | Reserved for future use |
| *int* | 0 if this is a triangle strip Patch, 1 if this is a point cloud Patch |
| *int* | ViSTa file index of the `TextureRef` that identifies the texture image to use for geometry in this patch ($\geq 0$) |
| *int* | Number of `Index` arrays in the patch ($\geq 1$) |
| *int* | Total number of `Index` listed in the patch ($\geq 3$ for triangle strip patches, $\geq 1$ for point cloud patches) |

The total size of a Patch is the size of the `PatchHeader`, plus the size of the array of `IndexArrayLengths`, plus the sum of the sizes of the `Index` arrays. An explicit size field in the `PatchHeader` would be redundant because the size can already be computed quickly from the existing fields.

See Section 9.3 for a description of the semantics of triangle strip structure.

## 10.8 IndexArrayLength

| Type | Value |
|---|---|
| *int* | The number of `Index` fields in the corresponding `Index` array ($\geq 3$ for `Index` arrays in triangle strip Patches; $\geq 1$ for `Index` arrays in point cloud Patches) |

See Section 9.3 for a description of the semantics of triangle strip structure.

### 10.9 Index

| Type | Value |
|---|---|
| *int* | An index into the array of `Vertex` ($\geq 0$) |

See Section 9.2 for a description of indices.

## 11 Acknowledgments

Thanks to Jack Morrison for information that was helpful to the development of this specification.

## References

[1] Jack Morrison. *ASD Mesh File Format PRELIMINARY.* 6/1/01, via email.

[2] Jack Morrison. Private communication. 3/8/02.

[3] SGI. *OpenGL Performer Programmer's Guide, Chapter 17: Active Surface Definition.* 2001, SGI document number 007-1680-060. Freely available in electronic form from http://techpubs.sgi.com.