NOAH

Storing Audiological Measurements

HIMSA Packed Scan Standard

DataFmtCodeStd 501

Version 1.0

HIMSA II K/S

Preface

This is the one of a series of documents prepared by HIMSA II K/S. Its purpose is to present and specify standard data formats for the storage and exchange of measurement related data within the framework of NOAH-compatible measurement and fitting software.

The Hearing Instrument Manufacturers' Software Association A/S (HIMSA A/S) was founded at the beginning of 1993 by a group of hearing instrument manufacturers. It has been HIMSA A/S's mission to develop and market the NOAH software, and to make it a de facto standard for integrated hearing care software within the entire hearing industry.

The NOAH Fitting Framework is a software application that enables fitting and measurement software to share data on a common platform (NOAH). The fitting and measurement applications are provided by manufacturers who have signed a know-how licence agreement with HIMSA and thereby obtained the right to distribute the NOAH software, and to develop NOAH-compatible software applications, also referred to as modules.

Data format standards are a natural prerequisite for the ability to share data. Therefore, in co-operation with its licensees, HIMSA has prepared data format standards for Audiogram, REM/HIT, Loudness Scaling, Impedance, Otoacoustic Emission and Evoked Response Audiometry measurement types.

The various data standards are subject to revision twice a year by a committee consisting of manufacturers of audiological measurement equipment (AEM's). Based on input prepared by HIMSA, it will be the responsibility of this committee to approve both new standard documents and updates of existing standards. When needed the AEM Committee will meet on the Saturday following the end of the UHA Convention in Germany, i.e. in October, and on the Saturday following the end of the AAA Convention in the US, i.e. in April.

HIMSA also invites non-licensees to take part in the process of preparing and maintaining measurement data standards.

Document History

Version	Author	Date	Changed pages	Change
1.0	SP	2015-04-22	All	This is an extended version of format 500 and this document has been updated via direct input from 3Shape who are the authors of this standard

1. Introduction

1.1 A few words about the packed scan standard (HPS)

The HIMSA packed standard (HPS) was developed to provide a way of storing ear scans in a more efficient way than existing standards to reduce the amount of data needed to be transferred with transactions thereby saving bandwidth and space in the NOAH database.

This was accomplished by developing a standard where data and commands are mixed in the data block.

The standard operates with two main types of data. Vertices are used to describe points and facets are used to describe the area between adjacent points. Data and commands are further described later in this document. A necessity of this approach is that the storage of commands and data must follow the sequence in which the scan was created.

The standard contains three compression schemes.

Schema CB provides the strongest compression by offering the ability to switch between absolute three dimensional coordinates and relative coordinates. When relative coordinates are used as vectors to the next points, a smaller amount of data can be used to describe the vector thereby saving space. In addition it offers the ability to colorize and apply texture to the scan using the facets functions.

Schema CC provides a simpler lossless implementation with a lower compression and lacking the ability to colorize and texturize the scan.

Schema CA compression schema implements the same format as the CC compression schema. It use is intended for backwards compatibility.

A simplified description of the creation of an ear scan using the HPS format would be to say that you start at a fixed point and then build the scan point by point referencing other points nearby as you go. The result is a triangular mesh in three dimensions.

This document is written as a part of the documentation for software developers of the NOAH Framework Programming Interface.

Data can be exchanged across these interfaces among the NOAH modules. In this way data can be shared between different Hearing Instrument- and Audiological Equipment-manufacturers.

1.2 Contents

1. HPS Compression schema CA

The CA compression schema implements the same format as the CC compression schema. It use is intended for backwards compatibility.

2. PS Compression schema CB

Schema CB is used to compress geometry data however still accommodating the use of texture. The main motivation for this initial compression schema CB is to enable an impression scan to be compressed to a size that allows it to be quickly transferred using a low speed internet connection. Furthermore a compressed scan requires less storage and backup resources.

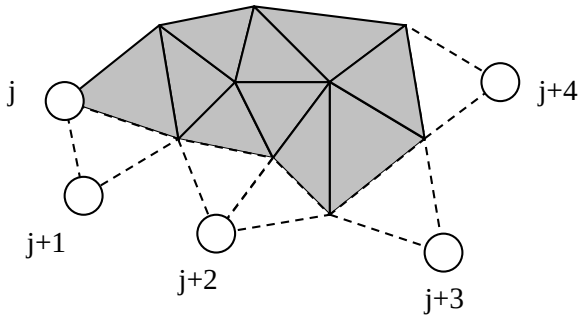
The CB compression schema is defined as follows:

Section	**Content**
<CB version=”1.0”>	
<CB_Vertices>	
	Vertices binary section (base64 encoded according to RFC 1521 by N. Borenstein and N. Freed) [1][command] [0][cluster header][cluster data] [0][cluster header][cluster data] . . . [0][cluster header][cluster data] [1][command] [1][command] [0][cluster header][cluster data] [0][cluster header][cluster data] . . . [0][cluster header][cluster data]
</CB_Vertices>	
<CB_Facets>	
	Facets binary section (base64 encoded according to RFC 1521 by N. Borenstein and N. Freed) [1][command] [0][cluster header][cluster data] [0][cluster header][cluster data] . . . [0][cluster header][cluster data] [1][command] [1][command] [0][cluster header][cluster data] [0][cluster header][cluster data] . . . [0][cluster header][cluster data]
</CB_Facets>	

</CB>	

In order to obtain a high degree of compression with the CB scheme, vertices must be packed in the vertex table according to time of usage in the mesh creation. Consider the following sketch:

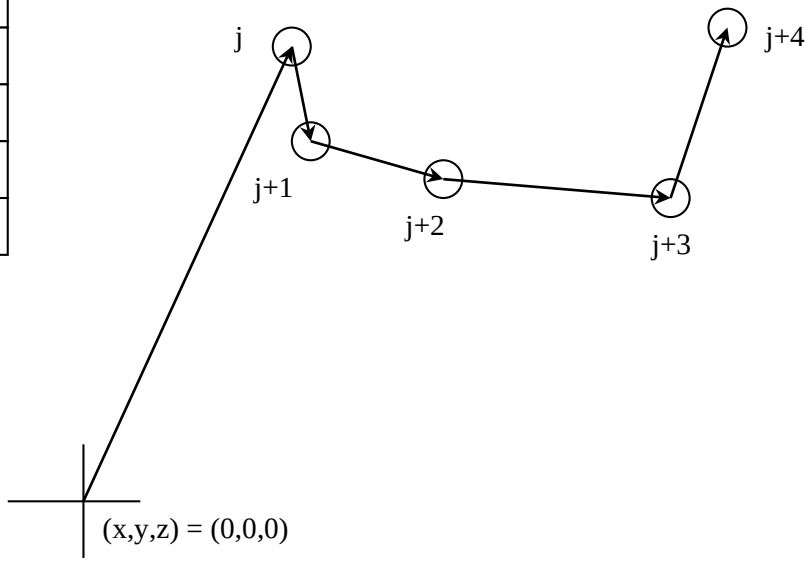
Vertex j
Vertex j+1
Vertex j+2
Vertex j+3
Vertex j+4



Hence vertices are packed in the order used to create the triangle mesh they span.

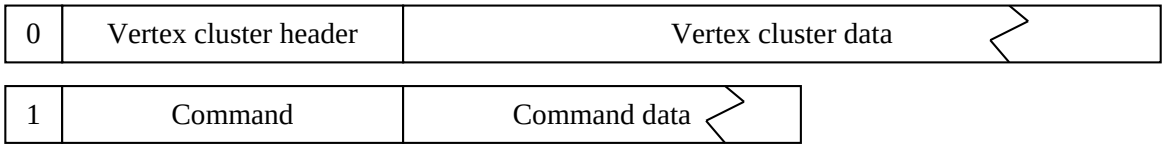
Further it's obviously important to code the vertices in matter that to reduce the overall size of the mesh. Vertex coordinates are coded using absolute and relative coordinate vectors. Based on the local neighbor assumption as sketched above, consecutive vertices can be coded using a minimum of information. Consider the sketch bellow.

Vertex j
Vertex j+1
Vertex j+2
Vertex j+3
Vertex j+4



Vertices binary section

The binary vertices section must contain all vertices of the triangle mesh. Vertices must be clustered together. Each cluster must start with a cluster header, specifying the format and size of the vertices in the following cluster. The vertex cluster data sections stores the vertex coordinates, incl. optional texture coordinates etc. Further it's possible to use special commands to change settings and parameters. A vertex cluster header must start with a 0 bit, whereas a special command must start with a 1 bit, as shown bellow.



To enable creation of the facet/triangle mesh, vertices must be globally numbered from 0 (the first vertex) to N (the last vertex), no matter the cluster they are stored in.

Vertex cluster header

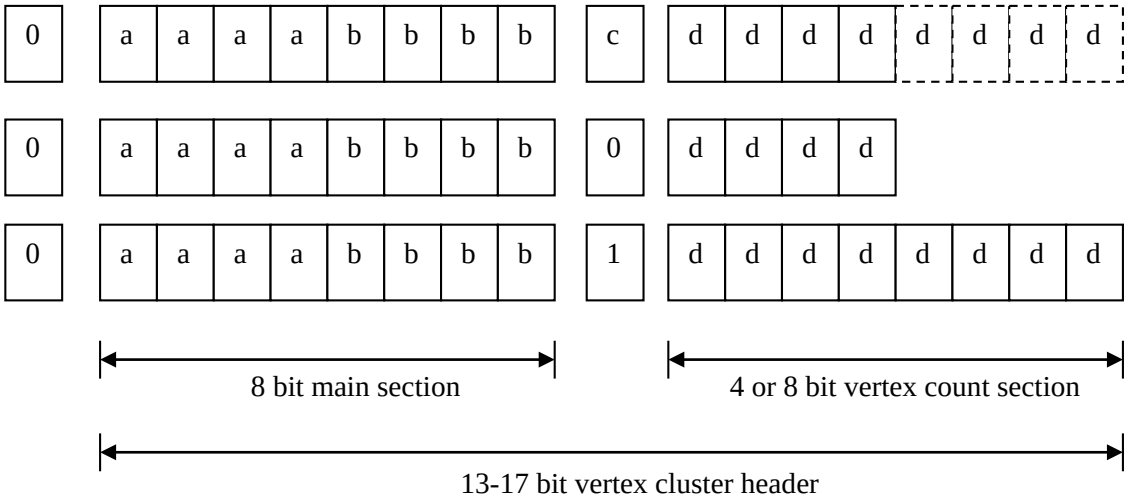
The vertex cluster header start with a fixed size (9bit) main section, that contains information bits in a well defined pattern (see bellow). It sets the x,y,z vertex coordinate format, i.e. how many bits used to store each x,y,z vertex absolute and relative coordinates. The main section is followed by a 4 or 8 bit vertex count section specifying the number of vertices in the following vertex cluster data section.

Bits in the vertex cluster header must follow the standard bellow:

Vertex cluster header, Main section (must always be present)		
Bit(s)	**Description**	**Valid values**
0-3 (a)	Number of bits used to express the first absolute x,y,z coordinates for the first vertex in the vertex cluster data section following the header. 0000: 8 bit representation (-128 to 127)*M 0001: 9 bit representation (-256 to 255)*M 0010: 10 bit representation (-512 to 512)*M 0011: 11 bit representation (-1024 to 1023)*M 0100: 12 bit representation (-2048 to 2.047)*M 0101: 13 bit representation (-4096 to 4.095)*M . . . 1111: 23 bit representation (-4,194,304 to 4,194,303)*M Where M is the coordinate vector multiplier (e.g. 0.005mm or 0.0002inch). M is set using a special command.	All
4-7 (b)	The number of bits used to express the absolute or relative x,y,z coordinates of the vertices following the first vertex in the vertex data section following the header. Relative or absolute according to mode set using the UseRelativeCoordinates and UseAbsoluteCoordinates. 0000: 4 bit representation (-8 to 7)*M	All

	0001: 5 bit representation (-16 to 15)*M 0010: 6 bit representation (-32 to 31)*M 0011: 7 bit representation (-64 to 63)*M 0100: 8 bit representation (-128 to 127)*M 0101: 9 bit representation (-256 to 255)*M . . . 1111: 19 bit representation (-262,144 to 262,143)*M Where M is the coordinate vector multiplier (e.g. 0.005mm or 0.0002inch). M is set using a special command.	
8 (c)	Number of bits used to specify the number of vertices in the following vertex cluster data section 0 : 4 bits (max 15 vertices in the data section) 1 : 8 bits (max 255 vertices in the data section)	0-1
Vertex cluster header, Vertex count section (size conditional on main header bit 8)		
9-12/16 (d)	Number of vertices in the following vertex cluster data section.	0-16/0-256

Using the vertex cluster header format as specified above the header looks as follows:



In the example shown in *Figure 1* the first absolute vector (j) is stored using 14 bit x,y,z coordinates. The following nine vertices are stored using relative 10 bit x,y,z coordinates.

```
Vertex cluster header = [0110 0110 0 1001]
One first vertex defined by three 14 bit absolute coordinates,
followed by eight vertices defined by three 10 bit relative
coordinates. In all, this cluster stores 9 vertices.

Vertex cluster data =   j:[xxxxxxxxxxxxxxyyyyyyyyyyyyyyzzzzzzzzzzzzzz]
                      j+1:[xxxxxxxxxxyyyyyyyyyyzzzzzzzzzz]
                      j+2:[xxxxxxxxxxyyyyyyyyyyzzzzzzzzzz]
                           ...
                      j+8:[xxxxxxxxxxyyyyyyyyyyzzzzzzzzzz]
```

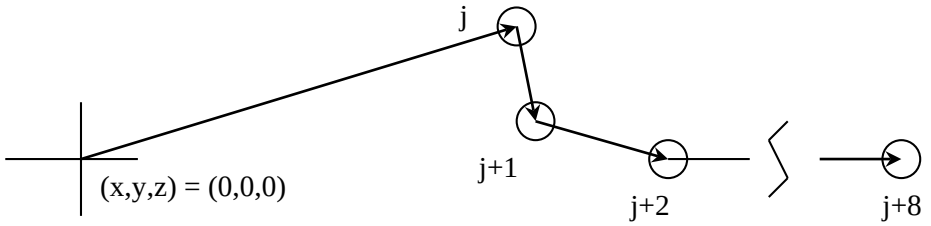


Figure 1: Absolute and relative vectors used to compress vertex x,y,z coordinates. In this case nine coordinates are stored using 295 bits or close to 37 bytes. In the example it's assumed that the mode has been set to UseRelativeCoordinates.

Vertex cluster data

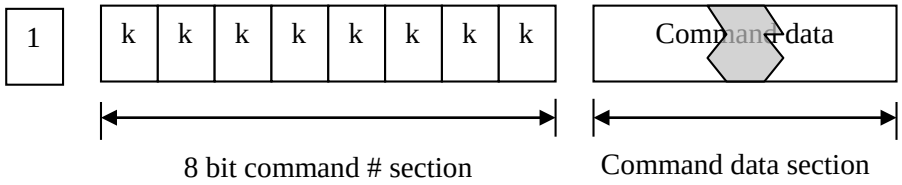
Following the vertex cluster header the vertices are stored according to the selected format. Vertices must be aligned according to the standard bellow:

Vertex in cluster data section, 3D coordinate section (must always be present)	
Size, bits	**Description**
8-23	1. Vertex, absolute X coordinate vector. Size according to header bits 0-3.
8-23	1. Vertex, absolute Y coordinate vector. Size according to header bits 0-3.
8-23	1. Vertex, absolute Z coordinate vector. Size according to header bits 0-3.
0/10-20	1. Vertex, texture image Xi coordinate. Size according to number of bits set using the command SetBitsPerTextureCoordinate, and only present if texture coordinates has been enabled.
0/10-20	1. Vertex, texture image Yi coordinate. Size according to number of bits set using the command SetBitsPerTextureCoordinate, and only present if texture coordinates has been enabled.
8-23	2. Vertex, absolute or relative X coordinate vector. Size according to header bits 4-7.
8-23	2. Vertex, absolute or relative Y coordinate vector. Size according to header bits 4-7.
8-23	2. Vertex, absolute or relative Z coordinate vector. Size according to header bits 4-7.
0/10-20	2. Vertex, texture image Xi coordinate. Size according to number of bits set using the command SetBitsPerTextureCoordinate, and only present if texture coordinates has been enabled.
0/10-20	2. Vertex, texture image Yi coordinate. Size according to number of bits set using the command SetBitsPerTextureCoordinate, and only present if texture coordinates has been enabled.
. . .	
8-23	n. Vertex, absolute or relative X coordinate … (see above)
8-23	n. Vertex, absolute or relative Y coordinate … (see above)
8-23	n. Vertex, absolute or relative Z coordinate … (see above)
0/10-20	n. Vertex, texture image Xi coordinate … (see above)
0/10-20	n. Vertex, texture image Yi coordinate … (see above)

Where n is the number of vertices defined by the header bits 9-16.

Special vertex cluster command

Using a 1 bit it's possible to give a special command. The command must follow immediately after the 1 bit as shown bellow.

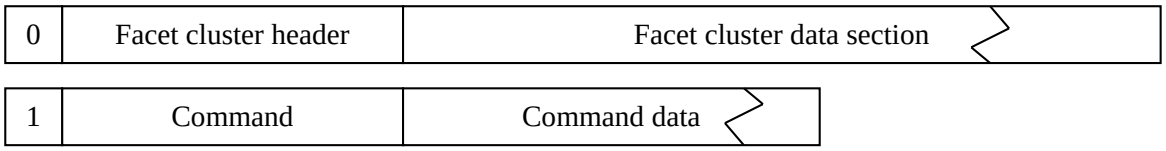


A limited number of commands are available and must be formatted using the standard bellow:

Vertex command section	
Command #	**Name – Description**
1 0000 0001 (default)	***UseRelativeCoordinates*** Use relative vertex coordinate vectors. The first vertex of a cluster must be stored using absolute x,y,z coordinates, the following vertices in the cluster must be stored using relative x,y,z coordinates.
2 0000 0010	***UseAbsoluteCoordinates*** Use absolute vertex coordinate vectors. All vertex coordinates after this command must be stores using absolute x,y,z values.
10 0000 1010 (default)	***DisableTextureCoordinates*** Vertex coordinates specified following this command cannot contain texture coordinates.
11 0000 1011	***EnableTextureCoordinates*** Vertex coordinates specified following this command must include texture coordinates using the specified bits per texture coordinate. The number of bits pr texture coordinate can be specified using the command SetBitsPerTextureCoordinate.
12 0000 1100	***SetTextureImage(ImageID:byte)*** Changes the current texture image. Image be defined in the Texture image section.
13 0000 1101	***SetBitsPerTextureCoordinate(n:byte) n [1-32]*** Texture coordinates following this command must be specified using n bits pr texture coordinate. n must be a number between 10 and 20 (both included). Default n=10.
20 0001 0100	***SetMultiplier(M:double)*** Change coordinate vector multiplier (M) Command data section then holds a 64 bit IEEE Double-Precision Floating Point Format (double),that M must be assigned to. Default M=0.01
30 0001 1110	***SetColor(R,G,B:byte)*** Change color. All following vertices should be colored according to the selected color. Command data section stores the three bytes that assigns the Red, Green and Blue levels for the following vertices. (24 bit standard color format). The default color must be set to R=0, G=0, B=0 (black vertices).

Facets (triangles) binary section

The facets binary section must contain all information needed to create the required triangle mesh. The mesh is constructed using a number of instructions. Like vertices are clustered together, facet creation instructions are clustered together. Each cluster must start with a cluster header, specifying the format used in the data section to create the mesh. The facet cluster data sections then holds instructions used to create the facet (triangle) mesh. As for the vertices it's possible to give a special command (e.g. to change the facet color).



Facet cluster header

The facet cluster header start with a fixed size (4bit) main section, that contains information bits in a well defined pattern (see bellow). The cluster header main section specifies the instruction set used to create the mesh in the following cluster data section. Further how many bits of the header reserved to specify the number of facets created by the instructions in the cluster data section. Finally the header specifies the number of facets created by the instructions in the following facet cluster data section.

Bits in the vertex cluster header must follow the standard bellow:

Facet cluster header, Main section (must always be present)		
Bit(s)	**Description**	**Valid values**
0 (a)	Facet pointers instruction set 0 : Reduced, mesh creation instruction set (2 bit) 1 : Full, mesh creation instruction set (4 bit)	0-1
1-2 (b)	Number of bits used to specify the number of facets created by instructions in the following facet cluster data section. 00 : 4 bits (max 15 facets in the data section) 01 : 8 bits (max 255 facets in the data section) 10 :12 bits (max 4095 facets in the data section) 11 :16 bits (max 65535 facets in the data section)	0-3
Facet cluster header, Facet count section (size conditional on header main section bit 1-2)		
Bit(s)	**Description**	**Valid values**
4-8/20 (c)	Number of facets created by instructions in the following facet cluster data section.	0-15 0-255 0-4095 0-65535

Facet cluster data

Following the facet cluster header the mesh creation instructions are stored in a format according to the selected instruction set. Two instruction sets are available: Reduced, mesh creation instruction set and Full, mesh creation instruction set. The instructions are defined in the following.

Reduced, mesh creation instruction set (2 bit)
The reduced mesh creation instruction set uses two bits pr. command, typically defining one new facet as a respond to each command.

Facet cluster data section, Reduced instruction set (header bit 0=0)	
Command	**Description**
0 00	***VertexList*** Create new facet from current edge, using the next vertex in the global vertex list. Following this, increase the global vertex list pointer by one. Assign current edge to edge being next in the current edge list, prior to creation of the new facet (see edge definitions).
1 01	***Previous*** Create new facet from current edge, using the vertex at the beginning of the previous edge (see edge definitions). Assign current edge to next as described above.
2 10	***Next*** Create new facet from current edge, using the vertex at the end of the next edge (see edge definitions). Assign current edge to next as described above.
3 11	***Ignore*** Ignore current edge, and assign current edge to the next edge in the edge list.

Full, mesh creation instruction set (4 bit)

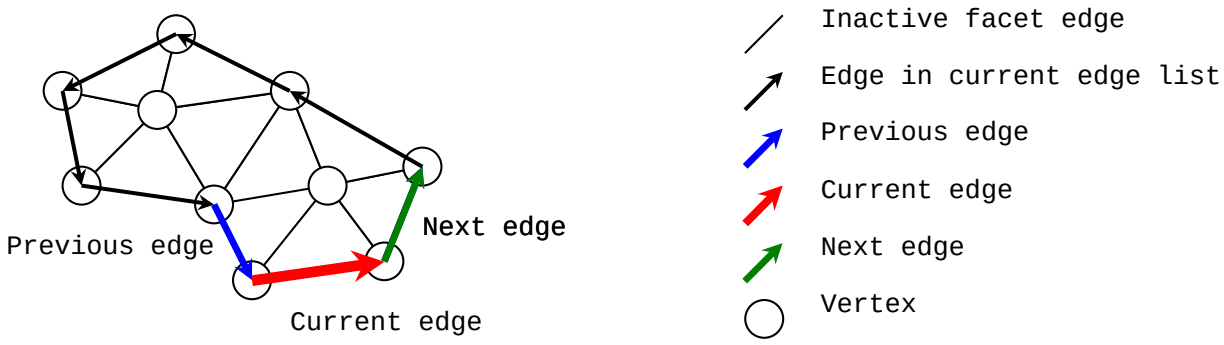
The full mesh creation instruction set uses four bits pr. command. Some commands must be followed by command parameters, such as absolute pointers to vertices in the global vertex list.

Facet cluster data section, Full instruction set (header bit 0=1)	
Command	**Description**
0 0000	***VertexList*** See 00 command in reduced instruction set.
1 0001	***Previous*** See 01 command in reduced instruction set.
2 0010	***Next*** See 10 command in reduced instruction set.
3 0011	***Ignore*** See 11 command in reduced instruction set.
4 0100	***Restart*** Create new facet using three next vertices in the global list. The cross product between vector from first vertex to second vertex and vector from first vertex to third vertex sets the direction of the surface front face. Following this instruction, a new edge list connecting the three specified vertices in the given order must be created. Further the current edge must be set to the edge spanned from the first to the second specified vertex.
5 0101	***Restart16(Vertex0, Vertex1, Vertex2 : 16BitWord)*** Create new facet using three specified vertices. Following the command, three 16 bit words must specify the absolute position of the vertices in the vertex list. The vertices must be specified so that the cross product between vector from vertex 0-1 and 0-2 is in the direction of the surface front face. (max 65,535 facets). Following this instruction, a new edge list connecting the three specified vertices in the given order must be created. Further the current edge must be set to the edge spanned from the first to the second specified vertex.
6 0110	***Restart32(Vertex0, Vertex1, Vertex2 : 32BitWord)*** Crate new facet using three specified vertices. As above, however using 32 bit word vertex pointers. (max 4,294,967,295 facets)
7 0111	***Absolute16(Vertex : 16BitWord)*** Create new facet from current edge using specified vertex. Following the command a 16 bit word must specify the position in the vertex list where to find the vertex.
8 1000	***Absolute32(Vertex : 32BitWord)*** Create new facet from current edge using specified vertex. As above, however using a 32 bit word to specify the position of the vertex in the global vertex list
9 1001	***Remove*** Remove edge from the current edge list, and assign current edge to the next edge in the list.

10 1010	***IncreaseVertexListPointer*** Increases the global vertex list pointer by one. This command is usually used on conjunction with Restart16 and Restart32, to align the vertex list pointer before using “VertexList” instructions.

Edge definitions

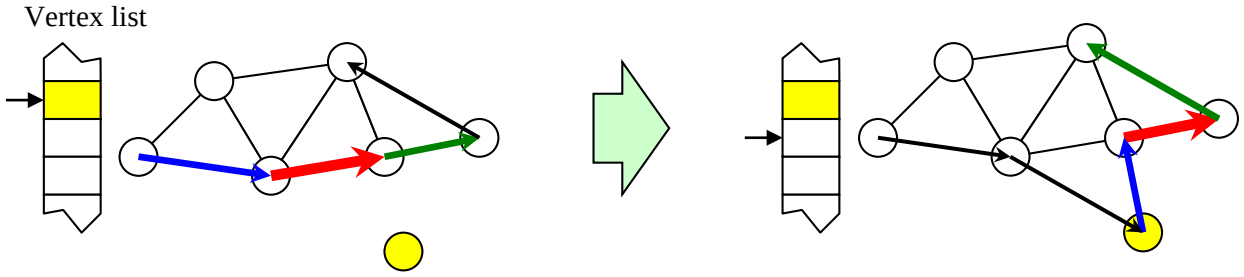
Following is the standard used to determine current edge, previous edge, next edge etc.



The instructions are described by examples in the following:

"VertexList" instruction

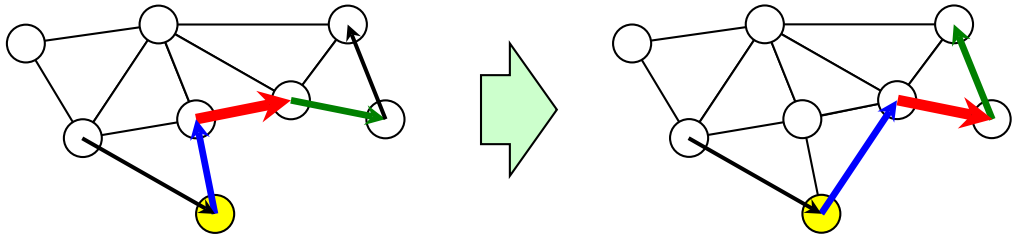
This instruction is used to expand the mesh from the current edge, using the next vertex in the global vertex list. The sketch bellow shows how it can be used to expand the mesh:



Mesh is expanded using the next vertex in the global vertex list (yellow) vertex. Following the instruction the global vertex list pointer is increased

"Previous" instruction

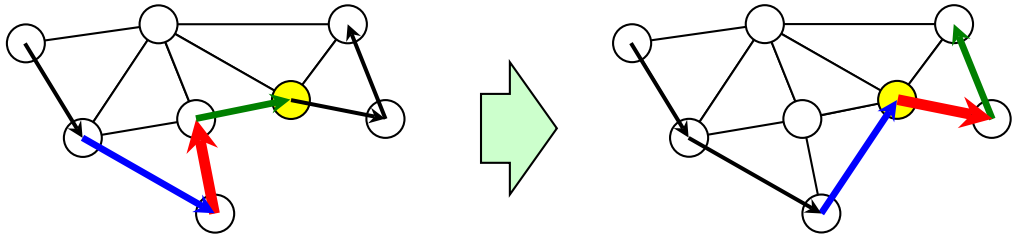
This instruction is used to expand the mesh from the current edge, using the vertex at the start of the previous edge.



Mesh is expanded using the yellow vertex.

“Next” instruction

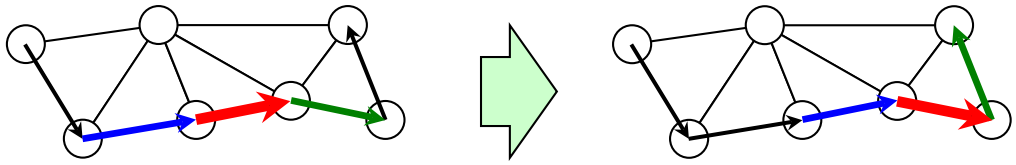
This instruction is used to expand the mesh from the current edge, using the vertex at the end of the next edge.



Mesh is expanded using the yellow vertex.

“Ignore” instruction

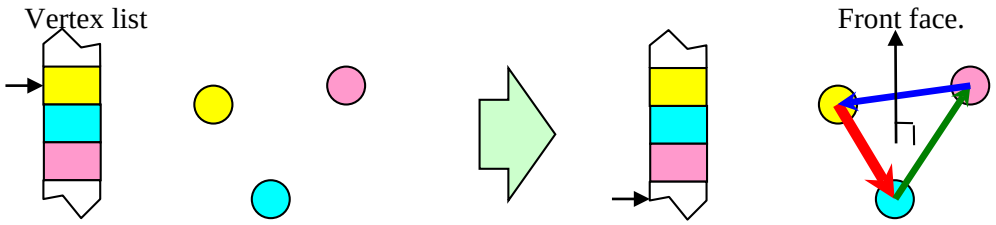
This instruction is used to ignore the current edge, and assign the current edge to the next edge in the edge list.



Current edge is assigned to the next edge, without any further expansion of the mesh.

“Restart” instruction

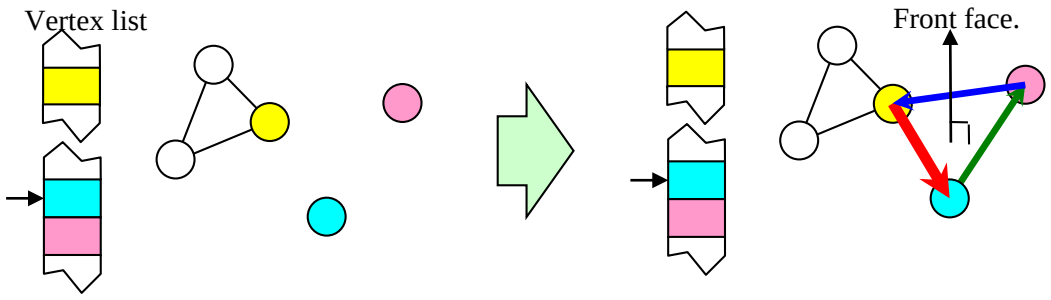
This instruction is used to create a new triangle and edge list using the tree next vertices in the global vertex list. Current edge is assigned to the edge from the first vertex to the second vertex.



Start of a new mesh or object using the three next vertices in the global vertex list. The global vertex list pointer is incremented by three.

“Restart16” and “Restart32” instructions

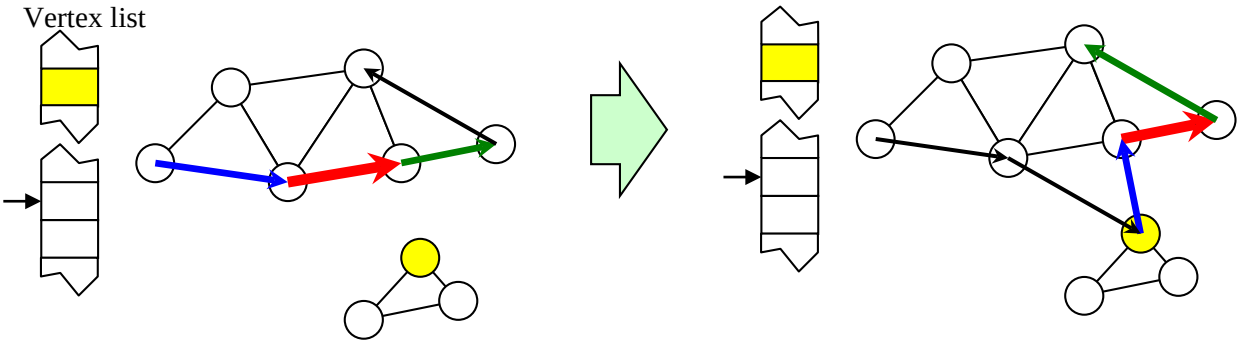
This instruction is used to create a new triangle and edge list using the three absolute indexed vertices from the global vertex list. Current edge is assigned to the edge from the first vertex to the second vertex.




```
Start of a new mesh or object using three absolute indexed vertices
from the global vertex list. The global vertex list pointer is
unchanged.
```

“Absolute16” and “Absolute32” instructions

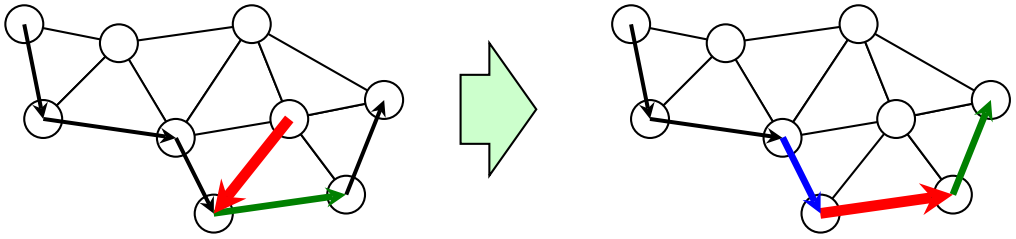
This instruction is used to expand the mesh from the current edge, using an absolute indexed vertex from the global vertex list.



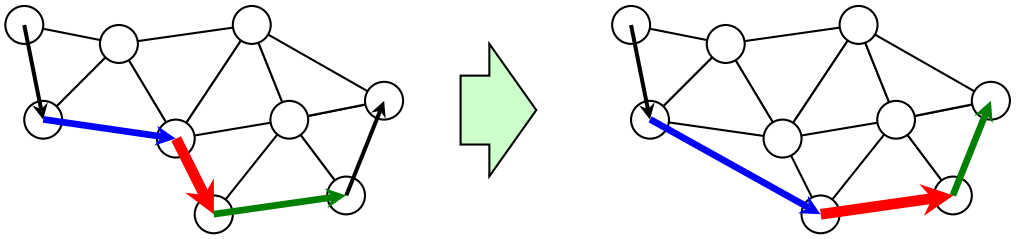

```
Mesh is expanded using global indexed vertex (yellow) from the
global vertex list. The global vertex list pointer is unchanged
```

“Remove” instruction

This instruction is used to remove edges from the edge list, thus avoiding further use of ignore commands. The two sketches bellow summarizes consequences of this instruction.



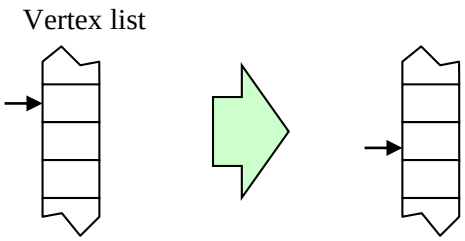
Case A: Current and previous edge are removed from the edge list.



Case B: Current edge is removed from the edge list.

“IncreaseVertexListPointer” instruction

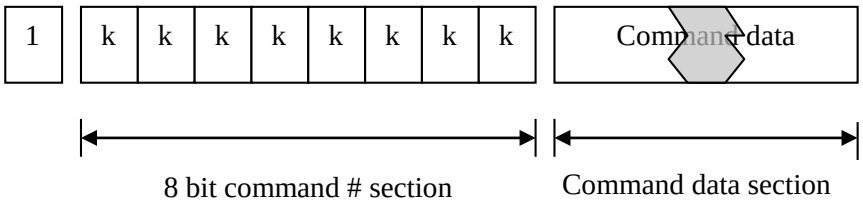
This instruction increases the vertex list pointer. It’s usually only necessary to use it in conjunction with “Restart16” and “Restart32” instructions, to assure a correct aligned vertex list pointer before using “VertexList” instructions.



Global vertex list pointer is increased by one.

Special facet cluster command

Using a 1 bit, it's possible to give a special command. The command must follow immediately after the 1 bit as follows:



A limited number of commands are available and must be formatted using the standard bellow:

Facet command section	
Command #	**Description**
0	
1	***SetColor*** If texture is not used, all facets/triangles created following this command must be colored according to the selected color. Command data section then holds three bytes that assigns the Red, Green and Blue levels for the following facets. (24 bit standard color format). The default color must be set to R=128, G=128, B=128 (native grey facets).

3. HPS Compression schema CC

The CC compression schema is similar to the CB compression scheme, however it accommodates for completely lossless compression and it's simple to implement compared to the CB compression scheme. Using the CC compression schema, vertices are stored using three standard 32 bit floats (IEEE Double-Precision Floating Point Format), one for each vertex coordinate (x,y,z). The mesh is stored/packed using the 4 bit Full instruction set as defined in the CB compression schema section (see Full, mesh creation instruction set (4 bit)). The main disadvantage of the CC compression schema is the lack of compression ratio, secondly it's not possible to store texture and store special commands, e.g. change color commands.

The CC compression schema is defined as follows:

Section	**Content**
<CC version="1.0">	
<CC_Vertices>	
	Vertices binary section (base64 encoded according to RFC 1521 by N. Borenstein and N. Freed) [1. vertex x,y,z coordinates (3x32 bit)] [2. vertex x,y,z coordinates (3x32 bit)] . . . [n. vertex x,y,z coordinates (3x32 bit)]
</CC_Vertices>	
<CC Facets>	
	Facets binary section (base64 encoded according to RFC 1521 by N. Borenstein and N. Freed) [1. mesh creation instruction (4 bit)] [2. mesh creation instruction (4 bit)] . . . [n. mesh creation instruction (4 bit)]
</CC_Facets>	
</CC>	